■ Hadoop、Spark两大框架，大数据技术大串讲

■ 技术干部、CIO、CTO、IT经理、初学者，大数据技术快速入门

大数据技术入门

Introduction to Big Data Technology

杨正洪 著



清華大学出版社

大数据技术入门

杨正洪 著

清華大學出版社
北 京

内 容 简 介

从 2015 年开始，国内大数据市场继续保持高速的发展态势，作者在与地方政府、证券金融公司的项目合作中发现，他们对大数据技术很感兴趣，并希望从大数据技术、大数据采集、管理、分析以及可视化等方面得到指导和应用帮助。因此编写了这本大数据技术的快速入门书。

本书共 12 章，以 Hadoop 和 Spark 框架为线索，比较全面地介绍了 Hadoop 技术、Spark 技术、大数据存储、大数据访问、大数据采集、大数据管理、大数据分析等内容。最后还给出两个案例：环保大数据和公安大数据，供读者参考。

本书适合大数据技术初学者，政府、金融机构的大数据应用决策和技术人员，IT 经理，CTO，CIO 等快速学习大数据技术。本书也可以作为高等院校和培训学校相关专业的培训教材。

图书在版编目（CIP）数据

大数据技术入门 / 杨正洪著. - 北京：清华大学出版社，2016

ISBN 978-7-302-44283-7

I. ①大… II. ①杨… III. ①数据处理 IV.①TP274

中国版本图书馆 CIP 数据核字（2016）第 164312 号

责任编辑：夏毓彦

封面设计：王　翔

责任校对：闫秀华

责任印制：

出版发行：清华大学出版社

网　　址：http://www.tup.com.cn，http://www.wqbook.com

地　　址：北京清华大学学研大厦 A 座　　**邮　　编：**100084

社 总 机：010-62770175　　**邮　　购：**010-62786544

投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn

质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn

印 刷 者：

装 订 者：

经　　销：全国新华书店

开　　本：190mm×260mm　　**印　　张：**19　　**字　　数：**486 千字

版　　次：2016 年 8 月第 1 版　　**印　　次：**2016 年 8 月第 1 次印刷

印　　数：1~3500

定　　价：59.00 元

产品编号：069895-01

前 言

我们生活在大数据时代，正以前所未有的速度和规模产生数据。数据资产正成为和土地、资本、人力并驾齐驱的关键生产要素，并在社会、经济、科学研究等方面颠覆人们探索世界的方法、驱动产业间的融合与分立。

大数据是用来描述数据规模巨大、数据类型复杂的数据集，它本身蕴含着丰富的价值。比如：在金融行业，企业和个人的一些信用记录、消费记录、客户点击数据集、客户刷卡、存取款、电子银行转账、微信评论等行为数据组合为金融大数据，他们利用大数据技术为财富客户推荐产品，利用客户行为数据设计满足客户需求的金融产品，利用金融行业全局数据了解业务运营薄弱点并加快内部数据处理速度，利用决策树技术进入抵押贷款管理，利用数据分析报告实施产业信贷风险控制，利用客户社交行为记录实施信用卡反欺诈，依据客户消费习惯、地理位置、销售时间进行推荐（精准营销）。不仅仅金融行业，政府部门会根据大数据分析结果来做预算，企业也会根据大数据来进行市场策略调整。

Gartner 指出，64%的受访企业表示他们正在或是即将进行大数据工作，然而其中一些企业却并不知道他们能够使用大数据做些什么。这正好印证了大数据领域的最主要的两个挑战：如何从大数据中获取价值以及如何定义大数据战略。这是本书首先需要解释的内容。

谷歌、Amazon、Facebook 等全球知名互联网企业作为大数据领域的先驱者，凭借自身力量进行大数据探索，甚至在必要时创造出相关工具。这些工具目前已经被视为大数据技术的基础，其中最知名的当数 MapReduce 与 Hadoop。Hadoop 是目前处理大规模结构化与非结构数据的首选平台，它提供了分布式处理框架与开发环境。MapReduce 是一种计算框架，它实现了将大型数据处理任务分解成很多单个的、可以在服务器集群中并行执行的任务，这些任务的计算结果可以合并在一起来计算最终的结果。在 Hadoop 问世以来的十年间，新的组件（如：Spark）层出不穷，极大地扩张了整个 Hadoop 生态圈。

大数据技术有别于传统数据处理工具和技术，而且大数据技术很难掌握，一般需要 1-2 年的反复尝试，在实际使用中解决了大量问题之后才能正确理解它。我们编写这本书的目的是，以硅谷大数据实战为基础，让读者略过那些不重要的大数据的细枝末节，通过实际的案例，帮助读者快速掌握大数据技术领域最能商用的大数据工具和软件平台，从而帮助读者轻松实施大数据方案。在本书中，我们将阐述如下最为硅谷所熟知的大数据相关技术：

- 框架：Hadoop、Spark。
- 集群管理：MapReduce、Yarn、Mesos。
- 开发语言：Java、Python、Scala、Pig、Hive、Spark SQL。

- 数据库：NoSQL、HBase、Cassandra、Impala。
- 文件系统：HDFS、Ceph。
- 搜索系统：Elastic Search。
- 采集系统：Flume、Sqoop、Kafka。
- 流式处理：Spark Streaming、Storm。
- 发行版：HortonWorks、Cloudera、MapR。
- 管理系统：Ambari、大数据管理平台。
- 机器学习：Spark MLlib、Mahout。

上面的列表也说明了，Hadoop 生态圈有几十个软件组成。这些软件提供了什么功能？到底在什么情况下使用什么软件？软件之间怎么组合使用？这些问题正是本书想要回答的。本书与市场上其他大数据书籍的区别是，我们不是专注某一个软件（比如：Spark），而是阐述整个生态圈中的主流软件，通过实例让你理解这些软件是什么，在什么场合使用，相互的区别是什么。如果我们把这几十个软件比喻成几十种厨房工具，那就是让你避免拿着菜刀去削苹果，或者拿着水果刀去剁肉。

除了阐述大数据的定义、前景和各类 Hadoop 发行版之外，本书主要是按照大数据处理的几个大步骤来组织内容的。

（1）大数据存储：探究 HDFS 和 HBase 作为大数据存储方式的优劣。

（2）大数据访问：探究 SQL 引擎层中 Hive、Phoenix、Spark SQL 等组件的功能，并阐述了全文搜索的 ElasticSearch，也探究了 Spark 的高速访问能力。

（3）大数据采集：大数据的采集是指接收各类数据源（比如：Web、行业应用系统或者传感器等）的数据。大数据采集的主要特点和挑战是导入的数据量大（每秒钟的导入量经常会达到百兆，甚至千兆级别）、并发数高和数据源的异构。采集端可能会有很多数据库（或文件），有时需要在导入基础上做一些简单的清洗和预处理工作。在这个部分，我们探究了 Flume、Kafka、Sqoop 等技术，也探究了如何使用 Storm 和 Spark Streaming 来对数据进行流式计算，来满足部分业务的实时和准实时计算需求。

（4）大数据管理：探究数据模型、安全控制、数据生命周期等数据管理内容。

（5）大数据的统计和分析：探究了如何利用分布式计算集群来对存储于其内的海量数据进行统计分析，重点探究了机器学习和 Spark MLlib，也阐述了多种分析算法。

参加本书编写的同志还有：余飞、邵敏华、欧阳涛、杨正礼、王娜、李祥、刘毕操、彭勃、李招、张剑、杨磊等人。由于我们水平有限，书中难免存在纰漏之处，敬请读者批评指正。杨正洪的邮件地址为 yangzhenghong@yahoo.com。

杨正洪

2016 年 5 月 于 San Jose

目　录

第 1 章 大数据时代

从 20 世纪开始，政府和各行各业（如：医疗、网络、金融、电信）的信息化得到了迅速发展，积累了海量数据。在这些数据当中，87%以上都是非结构化数据。虽然国内的各类数据中心已经有足够的硬件设施来存储这些数据，但是，如何让这些海量数据产生最大的商业价值，是目前面临的挑战之一。还有，由于数据的增长速度越来越快，数据量越来越大，传统的数据库或数据仓库很难存储、管理、查询和分析这些数据，如何在软件层面实现 PB 级乃至 ZB 级的海量数据存储和分析是目前面临的挑战之二。大数据（Big Data）技术就因此而生，并成功地解决了这两个挑战。以大数据的采集、整理、存储、管理、挖掘、共享、分析、反馈、应用为核心，最终实现智慧城市。根据 IDC 预测，2016 年的全球大数据市场规模将达到 230 亿美元。

1.1 什么是大数据

大数据不是一项单一的技术，而是一个概念，是一套技术，是一个生态圈。大数据技术和专业术语多达几十个，记录了大数据从炒作到成熟并进入主流应用的过程。数据科学家、预测分析、开放政府数据，都属于大数据范畴。大数据技术也逐渐变得越来越复杂。政府和企业希望从自己的数据中获得更多的信息，软件厂商希望将“大数据解决方案”融入公司的产品之中。在大数据软件公司的助推下，政府和企业已经有能力利用廉价的服务器、开源技术和云计算来进行开销不大的大数据部署。

对于什么是“大数据”，不同的研究机构从不同的角度给出了不同的定义。Gartner 认为：“大数据是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力的海量、高增长率和多样化的信息资产”。麦肯锡认为：“大数据指的是大小超出常规的数据库工具获取、存储、管理和分析能力的数据集。但它同时强调，并不是说一定要超过特定 TB 值的数据集才能算是大数据”。根据维基百科的定义，“大数据是指无法在可承受的时间范围内用常规软件工具进行捕捉、管理和处理的数据集合”。IDG 认为：“大数据一般会涉及 2 种或 2 种以上数据形式，它要收集超过 100TB 的数据，并且是高速实时数据流；或者是从小数据开始，但数据每年会增长 60%以上”。

从客户的角度来看，大数据技术的战略意义不在于拥有多么庞大的数据信息，而在于对这些含有意义的大数据进行专业化处理，从中获得商业价值。比如，以色列已经把所有政府部门的视频整合到一个大数据管理平台上，并在这个平台上开发了一套智慧安防系统。在这个系统上，只要把某一个人的人脸或人的主要特征数据输入系统，就能从海量的监控记录中查出同那个人相关的视频片段，并自动变成一个有时间顺序的片子。

随着以云计算、大数据、物联网等为代表的新一代信息技术的发展和应用，世界经济进入了大转型时代，主要发达国家以及国内发达省市都紧盯紧跟这一轮产业变革，试图抢占未来经济发展先机。大数据是一种产业，这种产业实现盈利的关键在于提高对数据的“加工能力”，通过“加工”实现数据的“增值”，完成“数据变现”。这种加工能力体现在技术上就是大数据分析。简言之，从各种各样类型的数据中，快速获得有价值信息的能力，就是大数据技术。大数据最核心的技术就是在于对于海量数据进行采集、存储、管理和分析。

1.2 大数据的四大特征

大数据具有 4 V 特征，即 Volume（数据体量大）、Variety（数据类型繁多）、Velocity（数据产生的速度快）、Value（数据价值密度低）。

Volume 指的是数据体量巨大。比如，一家 3 甲医院的影像数据（这包括 CT、B 超、X 光片、胃镜、肠镜等）可能就是几百个 TB，全国的医疗影像数据超过 PB 级别，接近 EB 级别。全球数据已进入 ZB 时代，IDC 预计 2020 年全球数据量为 40ZB。

Variety 指的是数据类型繁多。这可分为结构化数据、半结构化数据和非结构化数据。结构化数据，即行数据，存储在数据库里，可以用二维表结构来逻辑表达数据，比如企业财务系统、医疗 HIS 数据库、环境监测数据、政府行政审批等等。非结构化数据，一般存储在文件系统上，比如视频、音频、图片、图像、文档、文本等形式。典型案例有：医疗影像系统、教育视频点播、公安视频监控、国土 GIS、广电多媒体资源管理系统等应用。半结构化数据，介于完全结构化数据（如关系型数据库、面向对象数据库中的数据）和完全无结构的数据（如声音、图像文件等）之间的数据。比如邮件、HTML、报表等等，典型场景如邮件系统、教学资源库、档案系统等等。非结构化与半结构化数据的增长速率大于结构化数据，超过 80%的数据是非结构化数据。IDC 的报告显示，目前大数据的 1.8 万亿 GB 容量中，非结构化数据占到了 80%~90%，并且到 2020 年将以 44 倍的发展速度增加。非结构化数据比例不断升高，这些数据中蕴含着巨大的价值。

Velocity 是指大数据往往以数据流的形式动态、快速地产生，具有很强的时效性。数据自身的状态与价值也往往随时空变化而发生演变（这些数据往往包括了空间维、时间维等多种数据）。比如，环境监测中的水质和空气质量数据、高速路卡口的视频监测数据等。

Value 是指数据已经成为一类新型资产，蕴藏着大价值。大数据的价值密度低，需要通过专业的技术手段进行挖掘。只有对其进行正确、准确的分析，才会带来很高的价值回报。比如，电

视机顶盒的频道切换数据，各大电视台分析其中的数据，从中准确判断观众的喜好，以推出更加符合观众口味的节目。

大数据并非总是说有数百个 TB 才算得上。根据实际使用情况，有时候数百个 GB 的数据也可称为大数据，这主要要看它的其他维度，也就是速度或者时间维度。假如能在 1 秒之内分析处理 300GB 的数据，而通常情况下却需要花费 1 个小时的话，那么这种巨大变化所带来的结果就会极大地增加价值。所谓大数据技术，就是至少实现这四个判据（特征）中的几个。

1.3 大数据的商用化

大数据是传统的架构、传统的技术无法解决的数据处理问题。Hadoop 的出现，解决了大数据的快速存储和读取，也为我们提供了大数据分析的众多工具，但是，对于大数据商用而言，这并不够！因为大数据的名字有“大”，所以很多人把重点集中在了数据的容量上，简单地认为数据量是最大的问题。实际上大数据除了数据量的问题外，还会把信息管理的各项需求都推向极致（如图 1-1 所示）。

大数据质量管理（如：数据有效性、有效期限、相关性）

大数据访问的权限管理和控制（如：敏感性分级、共享协议、数据模型）
大数据基本问题（4V）

图 1-1　大数据商用需求

最下面的这一层是大数据基本的问题，包括大数据量、多样性、高速和低价值。解决这四个方面的问题只是大数据商用化的基础，这只是支撑起大数据的平台。Hadoop 技术很好地解决了这一层的问题。Hadoop 也被认为是下一代 IT 架构的基础，Hadoop 系统将逐步替换以关系型数据库为基础的传统系统。

中间这一层是关于访问权限的问题。数据的敏感性是一个很基础的问题，但是现有的 Hadoop 技术还没有对数据的敏感性提供可行的解决方案。那些提供大数据解决方案的 IT 企业不仅仅要关注大数据的 4V 量化指标，还需要把注意力放在“数据敏感性分级”上。国内超过 80% 的数据在政府的系统内。如果我们的大数据解决方案没有给政府数据提供诸如敏感性分级的权限管理机制，那么，政府是很难往前迈一步的。比如：公安、税务、工商等各部门的数据在一个平台上所产生的访问控制问题。共享协议是指数据将会以什么形式，通过什么样的接口实现数据交换，这是大数据的重点问题之一。数据交换的所有的方式都是以标准的协议来支持，因为在大数据的时代，数据的来源本身是多样性的，数据的格式甚至是无法管理的，很多的数据是来自于企业的外部，来自于互联网的提供商。到底如何通过这些协议和统一数据模型自动化地将数据放到

大数据平台上来，这是一个很严重的问题。Hadoop 本身并没有技术工具来解决这些方面的问题。

最上面一层是有关大数据质量的管理。数据本身是一种资产，资产质量怎么来衡量，我们如何确保数据的质量。这个也是我们在实施大数据商用上需要考虑的一个问题。质量管理是传统的数据管理里非常重要的一个方面，这包括数据的有效性和有效期限。Hadoop 本身并没有技术工具来解决这些方面的问题，但是我们需要相应的大数据工具和技术来解决这些问题，这就是我们下面阐述的大数据管理平台的作用。除了提供大数据质量的管理，这个管理平台还提供上述的大数据访问的权限管理等功能。

如图 1-2 所示，从用户的角度，从大数据平台的功能性的角度来看，我们把大数据平台细分为三个平台：大数据云平台、大数据管理平台和大数据应用（分析）平台。大数据采集（也叫数据交换和共享）包含在大数据管理平台之中。

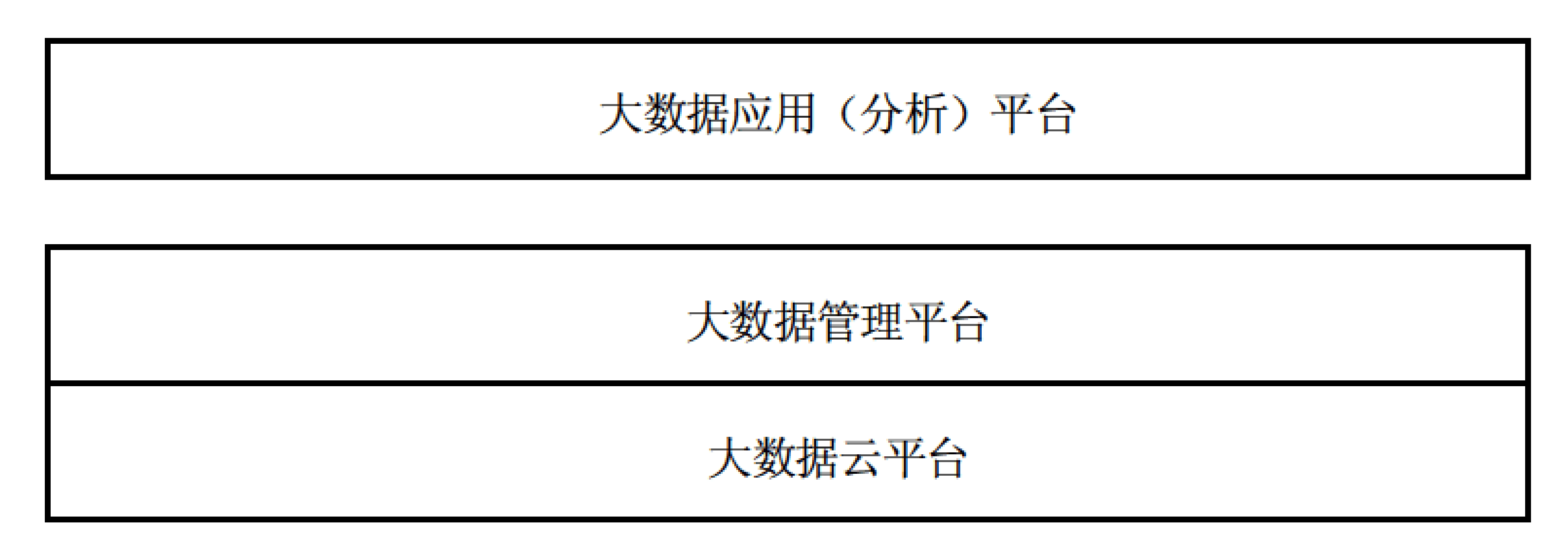


图 1-2　大数据建设总体架构图

大数据云平台是集约化建设的 IT 基础设施层，为大数据处理和应用提供统一的基础支撑服务；大数据管理平台是数据资源层，为大数据应用提供统一数据采集、分析和处理等支持服务；大数据应用平台是业务应用层，为大数据在各领域的应用提供综合服务。从逻辑结构上看，很多大数据应用需要基于大数据管理平台，而 Hadoop 技术只能完成大数据的底层功能，即：大数据的快速采取、存储和读取，所以 Hadoop 是大数据管理平台的基础。正是因为 Hadoop 缺少相应的数据管理技术和工具的支持，上述的一些非常基本的商用问题到现在还没有解决，这就凸显了大数据管理平台的重要性。只有提供了统一的大数据管理平台，数据的集成尤其是跨行业、跨不同的部门、跨各种技术的集成才能成为可能。整个大数据应用的架构必然是构建在一个大数据管理平台之上，这才可能实现大数据应用的大规模商用和普及，而不应该只是基于裸露的 Hadoop。

大数据顾名思义数据量庞大。在大数据时代，企业的数据不仅仅有传统的结构化数据，还有各类非结构化数据。结合对数据吞吐量的合理设计，将这些数据采集到大数据平台应该不会是很难的事情。比较难的是数据的转换、协调、确保不同数据源之间的一致性、检查数据的质量，这些是大数据采集中比较难实施的部分，而且在这些方面我们可用的自动化工具较少。

国内的大数据软件企业基本上都处于相对初级的阶段。很多新兴的公司提供 Hadoop 的发行版本的安装和配置，并针对 Hadoop 提供了一些定制化的应用。国内大数据软件产品和技术就是处于这么一个刚刚开始的状态。数据访问、安全、隐私、归档等，对数据管理来说，一些非常重要的、甚至于可以说是非常致命的需求，到现在仍然没有足够的解决方案。我们必须重点关注有

关数据管理的问题，因为这可能是大数据商用解决方案中的一个最薄弱的环节。

1.4 大数据分析

大数据平台可以存储所有类型的数据。从简单的文件存储，到不强调一致性的非关系型数据库存储。得益于自身基础设计理念，大数据平台可以无限扩展。如果大数据平台在云端运行维护，那么它的灵活性将更强。从概念上讲，存储数据是大数据应用中最易于实现的部分。

光有大数据还不够。那么，在大数据平台上存储了足够多的数据后，我们该怎么将其加以利用呢？分析大数据，并将分析结果应用于决策中才是最重要的事情。预测分析（predictive analytics）是大数据分析领域中的一个常用模式，它通过分析采集的数据来预测未来的行为或趋势。它根据事物的过去和现在估计未来，根据已知预测未知，从而减少对未来事物认识的不确定性，以用来指导我们的决策行动，减少决策的盲目性。在大数据分析领域，预测分析常常与预测模型、机器学习和数据挖掘有关。对于一个政府部门而言，通过预测分析来精准把握政府工作的重点。比如：云升科技帮助湖州市公安局分析来自各个渠道的海量群众诉求，预测下个月的警务工作热点，从而帮助湖州市公安局合理安排警力，最终实现民意引领警务。美国的医疗决策支持系统基于预测分析来判断某些人得某些疾病的风险，并基于当前的健康状态给出最正确的医疗决定。国内的很多金融企业通过预测分析来实现业务的风险控制。比如：某银行分析其客户的消费数据和基本数据，从而预测该客户的信用卡和贷款的偿还能力。环保部门用数据决策，利用环保大数据综合研判，制定环境政策措施，预警环境风险，提供环境综合治理科学化水平。

除了预测分析，还有关联分析。关联分析的目的在于，找出数据之间内在的联系。比如，购物篮分析，即消费者常常会同时购买哪些产品（例如游泳裤、防晒霜），从而有助于商家的捆绑销售。

1.5 大数据与云计算的关系

大数据 IT 架构的基本特征，首先必须是可以横向扩展的，因为单点的技术无法承受大数据的要求。既然实现了通过横向扩展的架构来提升性能，这就没有必要在每个节点上花费太多的钱。它的高可用性是通过软件设计和架构设计来实现的，而不是通过传统的高性能、高可用性的高端硬件设备来实现的。所以，从技术上看，大数据与云计算的关系就像一枚硬币的正反面一样密不可分。大数据必然无法用单台的计算机进行处理，必须采用分布式架构。而云计算的分布式处理、云存储和虚拟化技术为大数据提供了 IT 基础，保证了大数据应用的高效运行。

正如图 1-2 所示，未来的趋势是，云计算作为计算资源的底层，支撑着上层的大数据处理，而大数据的发展为云计算的落地找到了更多的实际应用。大数据和云的融合将是重大的趋势，这

两个技术是相辅相成的关系。从目前部署上看，大概 70%的大数据系统在企业的内部，其中一些是在企业的私有云上。当然，大数据也可以部署在公有云上，比如：AWS、Azure 和 GCP（Google 云平台）提供了 Hadoop 基础设施服务，GCP、IBM、Oracle、Pivotal CF、Microsoft Azure 和 Qubole 提供了托管式 Hadoop 服务（平台即服务）。有些 IT 企业也提供了云上的分析即服务的功能。还有一种大数据系统部署是在一体机上，大概占到 5%左右的市场份额。提供 Hadoop 一体机的厂商有 DELL、EMC、ORACLE、Teradata 和惠普。

我们以 Amazon 的 AWS 为例来看一个云计算和大数据结合的实例。AWS 总体上成熟度很高， Netflix、Pinterest、Coursera 都在使用 AWS。如图 1-3 所示，S3 是简单面向对象的存储，DynamoDB 是对关系型数据库的补充，Glacier 对冷数据做归档处理，EC2 就是基础的虚拟主机，Elastic MapReduce（EMR）直接打包 MapReduce 来提供计算服务，使用 EMR 可以按需组建一个由节点组成的集群。这些集群用于 Hadoop 的安装和配置。Amazon 提供了非常类似 Kafka 的服务，称之为 Kinesis。它同时作为使用 EC2 进行分布式流处理的基础。

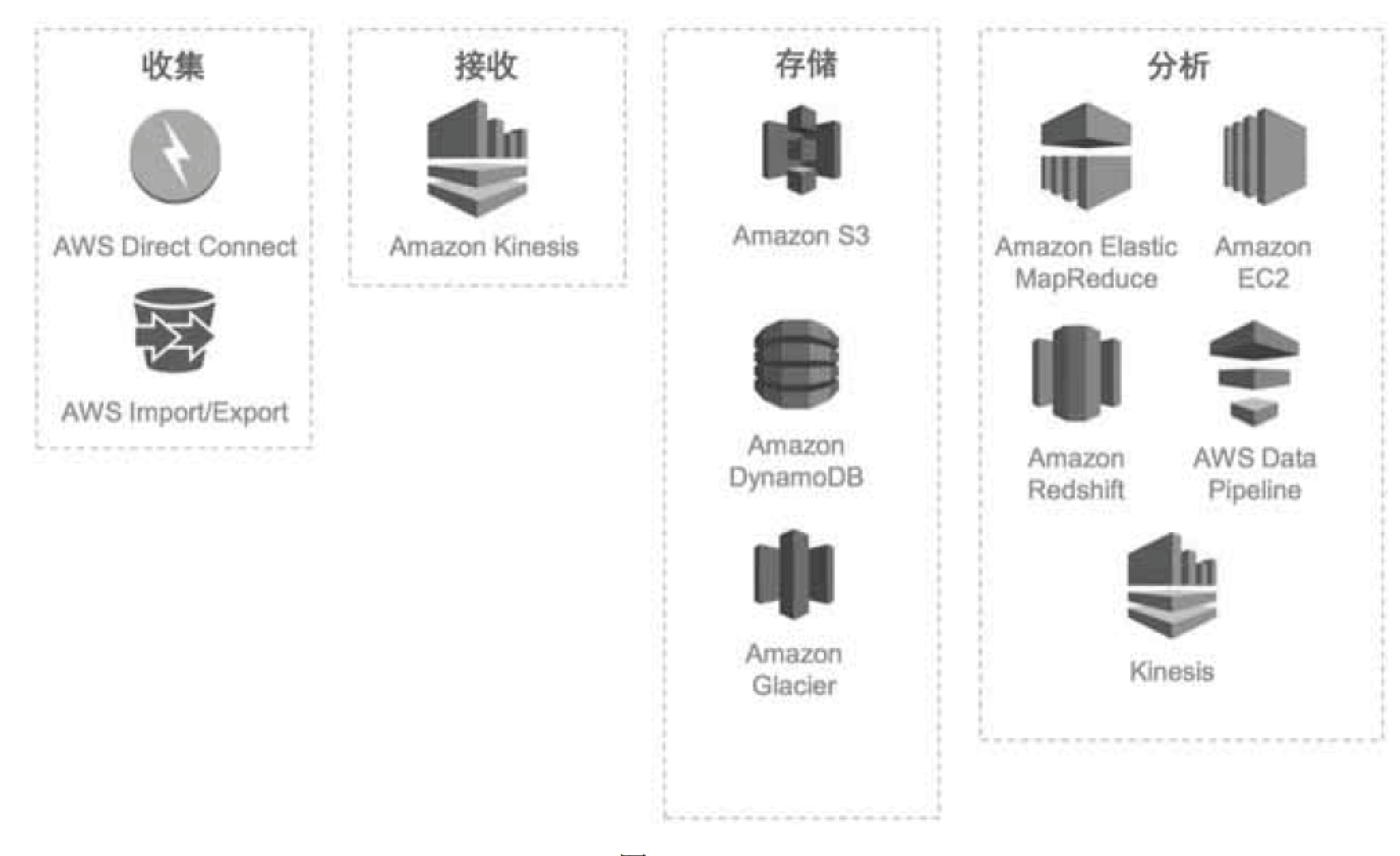


图 1-3　AWS

1.6 大数据的国家战略

斯诺登效应导致政府和企业对信息产品的安全性非常关注。政府和大型企业已经将关注重点转移向开源软件上，因为开源软件被认为是更加透明和安全的。这是以 Hadoop 为代表的开源软件在国内发光发热的大机遇。另外，2015 年 8 月 19 日，国务院常务会议通过了《关于促进大数据发展的行动纲要》。会议强调：一要推动政府信息系统和公共数据互联共享，消除信息孤岛，

加快整合各类政府信息平台，避免重复建设和数据打架；二要顺应潮流引导支持大数据产业发展，以企业为主体、以市场为导向，加大政策支持，着力营造宽松公平的环境，建立市场化应用机制，深化大数据在各行业的创新应用，催生新业态、新模式，形成与需求紧密结合的大数据产品体系，使开放的大数据成为促进创业创新的新动力；三要强化信息安全保障，完善产业标准体系，依法依规打击数据滥用、侵犯隐私等行为。让各类主体公平分享大数据带来的技术、制度和创新红利。这个行动纲要明确指出推动政府大数据开放、共享和安全的重要性。

2016 年 3 月 17 日发布的“十三五”规划纲要中指出，实施国家大数据战略。把大数据作为基础性战略资源，全面实施促进大数据发展行动，加快推动数据资源共享开放和开发应用，助力产业转型升级和社会治理创新。加快政府数据开放共享，全面推进重点领域大数据高效采集、有效整合，深化政府数据和社会数据关联分析、融合利用，提高宏观调控、市场监管、社会治理和公共服务精准性和有效性，依托政府数据统一共享交换平台，加快推进跨部门数据资源共享共用。加快建设国家政府数据统一开放平台，推动政府信息系统和公共数据互联开放共享。制定政府数据共享开放目录，依法推进数据资源向社会开放。统筹布局建设国家大数据平台、数据中心等基础设施。

1.6.1　政府大数据的价值

对大数据企业来说，目前遇到的发展机遇已经十分清楚。行动纲要的三个关键词的出发点和落脚点都指向政府大数据，关于政府大数据的开放共享，关于政府大数据的研究与应用，关于政府大数据的示范效应。那么我们不禁要问，政府大数据的价值究竟何在？为什么政府大数据更有价值？这就要从数量和质量两个层面说起。

就数量而言，或许有人会问，政府数据的数量能比得过 BAT 吗？表面上看，百度、阿里和腾讯都分别拥有数以亿计的用户量，但这与政府大数据相比，不是一个量级，可谓小巫见大巫。阿里巴巴的数据容量在 100PB 左右，而仅一个北京市政府就拥有几百个 PB 的数据容量，相当于几个阿里巴巴。这还仅仅是一个北京市政府。中国有几百个城市，几千个行政县。当前，中央和省级政务部门的电子政务覆盖率已经达到 70%。粗略估算，全国政府大数据加起来至少也该有数百甚至上千个阿里巴巴的体量。

至于政府大数据的质量，我们也可以通过和 BAT 企业对比来说明。比如百度，他拥有庞大的用户搜索记录，但这些数据较为单一，不进行关联应用毫无价值；腾讯的优势在于拥有数亿的 QQ 和微信用户量以及更庞大的社交数据，但这些数据目前仅局限于营销应用；阿里的交易数据似乎价值更高，但也只是局限在电商领域以及外延应用。换句话说，这三家 BAT 企业的短板共同点在于数据种类的单一化程度较高。政府大数据不同，它涉及工商、税务、司法、交通、医疗、教育、通信、金融、地理、气象、房产、保险、农业、环境等等领域，数据的种类繁多，关联性强、统计规格较为统一，便于应用处理。政府的数据事关百姓生活的方方面面，数据的利用价值也最高。

1.6.2 政府大数据的应用场景

各地政府都非常关注政府大数据。比如：为推进浙江省信息资源的整合开放和大数据产业发展，浙江省政府在 2015 年成立了浙江省数据管理中心。浙江省数据管理中心将拟定并组织实施大数据发展规划和政策措施；研究制定数据资源采集、应用、共享等标准规范；统筹推进大数据基础设施建设、管理；组织协调大数据资源归集整合、共享开放，推进大数据应用；组织协调大数据信息安全保障体系建设。浙江省政府成立大数据发展领导小组，负责大数据发展的政策制定、相关整合工作、数据开放共享等顶层设计职能。

中国对政务相关的政府大数据产业需求是非常的明显。由于全国各地区信息化发展的水平差异较大，政务信息化建设也存在着明显的区域差异性特征。国务院通过的《行动纲要》提出，“要推动政府信息系统和公共数据互联共享，消除信息孤岛，加快整合各类政府信息平台，避免重复建设和数据打架”，主要针对的就是政务数据的开放共享平台而言的。很多地市正在兴建公共信息服务平台，把全市政府资源数据集中存储和统一管理。一些省市已经设立了大数据管理局。

除了上述的智慧政务之外，中国政府大数据还主要应用于以下领域：智慧城市、公共服务、医疗、教育、交通、环境保护、能源等，这些领域大多涉及国民生活和城镇化进程。截至 2016 年初，中国的智慧城市试点已达 193 个，而公开宣布建设智慧城市的城市超过 400 个，投资总规模高达 5000 亿元。智慧城市的概念包含了智慧政务、智慧能源、智慧交通、智慧医疗、智慧环保等多领域的应用，而这些都要依托于大数据，大数据产业是“智慧”的源泉，是智慧城市的推手。

有专家曾将目前政府大数据发展现状看作是城市建设自来水管道系统时期。每个城市只建一套自来水供水系统，不可能建第二套，所以，快速布局政府大数据云平台和大数据管理平台是大数据基础建设的第一步，也是赢得政府大数据市场的第一步。至于自来水（即：平台上所管理的数据）问题，那就是基础建设后顺理成章的事了。谁拿到了这个自来水管道建设权，谁就是未来的数据之王。还有，大数据基础建设是大数据行业发展的前期环节，等到基础环节铺设完善，自来水得以通畅流通了，那时政府大数据的价值才真正爆发出来，那就进入大数据商业应用的时期了。大数据管道建设涉及设计和技术等十分专业的工作，政府的策略是请专业公司提供大数据管理平台，以作为大数据管道和基石。

1.7 企业如何迎接大数据

大数据问题不单单发生在互联网等新的事物的数据上，有很多问题发生在企业的传统应用所产生的数据上。随着数据量的增长，现有的 IT 架构慢慢地不能满足其要求。也就是说，大数据一半是新的业务，另外一半是解决传统业务的性能问题和管理数据的成本问题。比如，中国移动的某个阅读基地，在数据库上的数据为几百亿行，单表在 10 亿行左右。这使得数据库系统经常接近崩溃的边缘，技术人员把大多数时间放在系统管理和维护上。从 2014 年开始，中国移动的阅读基地把数据系统移向 Hadoop 系统，从而彻底解决了大数据量所引起的问题。

大数据新的应用是一个补充，是一个创新应用，而不是去替换传统的应用。如今的数据是多种数据的混合体，它不能事先预知数据的格式和形态。实际上很多的数据可能不是由企业本身所拥有的，而是从外部收集或购买，这样的话，传统的应用就施展不开了。

大数据不应该只是 IT 部门的事情，而是全公司协同作战的事情。管理层可以从大数据中获得洞察做决策，运营部门可以根据数据分析结果来改善运营策略，市场部门可以从数据分析中来优化广告投放策略，甚至是客服部门也可以从数据分析结果中来优化自己的工作，更别提销售部门了，他们更需要大数据的支持。

大数据是个机遇，也是个挑战，它是一个用传统的技术方法无法解决的数据问题，这对于企业来说是一个挑战。企业要迅速接受大数据的概念，这不单单是从解决现有的 IT 问题的角度考虑，更多的应该从未来的新的利润增长点和新的竞争点的角度来考虑，应该采用非常积极的态度。企业要认识到大数据不是在现有架构上新增应用，而是彻底改变现有架构。实施大数据前确定每一步的投资规模，设立里程碑和阶段目标，了解其技术和商业不成熟性可能带来的失误和风险，避免陷入厂商的炒作陷阱。从 IT 部门的角度来看，数据的价值应该说是由业务部门来决定的，所以必须要充分了解业务的需求。

关于大数据，企业首先应该考虑的问题不是这些数据能为我赚多少钱，而是如果我不去整合内部和外部的数据、存储数据、分析数据，那么未来我会失去多少钱？我会比竞争对手落后多少？数据的整合不是一朝一夕的事情，而是需要经过一段时间的累积。有些数据是需要从其他渠道拿到。整合数据和数据分析本身就不是先有鸡还是先有蛋的问题，而是你不养鸡，你肯定就不会有蛋。在未来的竞争格局中，数据往往能发挥先发制人的作用和优势。

1.7.1　评估大数据方案的维度

对于企业而言，构建大数据平台，是个系统性的工程。企业可以选择以增量方式实现大数据解决方案。不是每个分析和报告需求都需要大数据解决方案。随着大数据技术的到来，我们会问自己：“大数据是否是我的业务问题的正确解决方案，或者它是否为我提供了新的业务机会？”“企业 IT 部门需要掌握哪些技能来理解和分析软件厂商的大数据解决方案？”　“现有企业数据和来自外部的数据的复杂性”　“哪些维度可帮助评估大数据解决方案的可行性?”。

为了回答上述这些问题，业内专业人士提出了以下多种维度来评估大数据解决方案的可行性。企业应该依据自身业务的特点，为每个维度分配一个权重和优先级。

- 数据整合和分析所带来的业务价值。
- 数据整合（无论是新来源的数据还是原有数据）后的数据治理考虑。
- 企业是否自己拥有大数据技术人员，厂商是否有足够的技术支持人员。
- 整个数据量。
- 各种各样的数据源、数据类型和数据格式。
- 生成数据的速度，需要对它处理的速度。
- 数据的真实性，或者数据的不确定性和可信赖性。

1.7.2 业务价值维度

许多企业想知道，大数据产品能否帮助他们找到业务机会。所以，业务价值维度是指通过大数据技术可以为企业获取哪些新业务或者解决哪些现有的问题？这需要确定和识别大数据的业务场景，并给出关键绩效指标。这包括研究竞争对手的行动，知晓客户在寻找什么。表 1-1 按照行业给出了一些大数据的应用示例。

表 1-1 分行业大数据的应用示例

行业	业务示例
金融服务	• 合规性和监管报告 • 风险分析和管理 • 欺诈检测和安全分析 • 客户忠诚度计划 • 信用风险、评分和分析 • 交易监管 • 异常交易模式分析
欺诈监测	欺诈管理可预测给定交易或客户账户遇到欺诈的可能性。大数据解决方案将会实时分析交易并预警，这对阻止欺诈至关重要。这些欺诈类型有： • 信用卡和借记卡欺诈 • 存款账户欺诈 • 技术欺诈和坏账 • 医疗欺诈 • 医疗补助计划和医疗保险欺诈 • 财产和灾害保险欺诈 • 工伤赔偿欺诈 • 保险欺诈（如：汽车保险）
网站	• 大规模点击流分析 • 广告投放、分析、预测和优化 • 社交图分析和概要细分 • 营销活动管理和忠诚度分析
公共领域	• 网络安全 • 合规性和监管分析 • 能耗和碳排放管理

（续表）

行业	业务示例
医疗健康	• 健康保险欺诈检测 • 营销活动和销售计划优化 • 品牌管理 • 患者护理质量和程序分析 • 医疗设备和药物供应链管理 • 药品发现和开发分析
运营商	• 客户流失预防 • 营销活动管理和客户忠诚度分析 • 呼叫详细记录（CDR）分析 • 网络性能和优化 • 移动用户位置分析
能源	每个电网包含监视电压、电流、频率和其他重要操作特征的复杂传感器。分析发电（供应）和电力消耗（需求）数据。分析智慧电表（也适用于水表）的数据
电商	• 推荐引擎：通过基于对交叉销售的预测分析来推荐产品 • 下一款最佳产品：结合预测模型和现有产品销售信息，确定下一款最佳产品
零售业	• 营销活动管理和客户忠诚度分析 • 供应链管理和分析 • 市场和用户细分 • 预测分析：在产品上架之前，预测对购买者重要的一些因素
广电	使用大数据来分析机顶盒数据。可以利用此数据来调整广告或促销活动
政府部门（以环保为例）	• 整合雾霾监测历史数据，同步集成气象、遥感、排放清单、环境执法数据，形成雾霾案例知识库，开展雾霾预测预警服务 • 利用环境违法举报、互联网采集等环境信息采集渠道，结合企业的工商、税务、质检等信息，开展大数据分析，精确打击企业未批先建、偷排漏排、超标排放等违法行为
其他行业	• 交友网站：分析各个成员之间的兼容性，给出合理的推荐 • 飞机和汽车（尤其是长途大巴）的预测性维护

1.7.3　数据维度

数据维度包括数据优先级维度、数据复杂性维度、数据量维度、数据种类维度、数据处理速度和数据可信度。

首先要为企业（或政府部门）的现有数据整理出一个编目（清单），用于识别内部的应用系

统中存在的数据以及从第三方传入的数据。如果业务问题可使用现有数据解决，那么就不需要使用来自外部的数据。有些客户有一些归档数据，分析归档数据来获得新的业务价值。在有些时候，包括日志文件、错误文件和来自应用程序的操作数据都是宝贵信息的潜在来源。

其次要确定数据复杂性是否在增长？数据复杂性的增长可能表现在数据量、种类、速度和真实性方面。然后要判断数据量是否已增长？如果满足以下条件，企业可考虑大数据解决方案：

- 数据大小达到 PB 和 EB 级，而且未来有可能增长到 ZB 级别。
- 数据量给传统系统（比如关系型数据库）的存储、查询、共享、分析和可视化数据带来挑战。

还有一点是，数据种类是否已增多？如果满足以下条件，那可能需要大数据解决方案：

- 数据内容和结构无法预期或预测。
- 数据格式各不相同，包括结构化、半结构化和非结构化数据。用户和机器能够以任何格式生成数据，例如：Microsoft Word 文件、Microsoft Excel 电子表格、Microsoft PowerPoint 演示文稿、PDF 文件、社交媒体、Web 和软件日志、电子邮件、来自相机的照片和视频、传感设备数据、基因组和医疗记录。
- 不断出现新的数据类型。

最后还要考虑的是，数据的增长和处理的速度。是否需要即时响应，是否需要实时处理传入的数据。对于数据是否值得信赖，如果满足以下条件，那么需要考虑使用大数据解决方案：

- 数据的真实性或准确性未知。
- 数据包含模糊不清的信息。
- 不清楚数据是否完整。

如果数据的量、种类、速度或真实性具有合理的复杂性，那么就采用大数据解决方案。对于更复杂的数据，需要评估与实现大数据解决方案关联的任何风险。对于不太复杂的数据，则应该评估传统的解决方案。

1.7.4 现有 IT 环境和成本维度

对于想要通过大数据分析获取业务价值的情况，我们还要考虑当前的 IT 环境是否可扩展。与企业 IT 部门沟通，询问以下问题，确定能否扩展现有的 IT 平台？

- 当前的数据集是否非常大，是否达到了 TB 或 PB 数量级？
- 现有的数据仓库系统是否包含所有数据？
- 是否有大量冷数据（人们很少接触的数据）未分析？可以通过分析这些数据获得业务价值吗？
- 是否需要丢弃数据，因为无法存储或处理它？
- 是否希望在复杂且大量的数据上执行数据探索？

- 是否希望对非结构化数据进行分析？

对于这些问题的回答，可以帮助企业判断是扩充现有数据仓库系统还是部署一套新的大数据平台软件。还有一点，我们要比较这两个方案的成本。扩展现有 IT 环境与部署大数据系统的成本和可行性取决于：

- 现有工具和技术。
- 现有系统的可伸缩性。
- 现有环境的处理能力。
- 现有平台的存储能力。
- 执行的治理和策略。
- 现有应用系统的异构性。
- 企业 IT 部门的技术能力（包括为此需要新招人员的成本）。
- 从新数据源收集的数据量和成本。
- 新业务的复杂性。

我们要考虑大数据工具和技术需要的基础架构、硬件、软件和维护的成本。大数据解决方案可以采用增量方式实现。明确地定义业务问题的范围，并以可度量的方式设置预期的业务收入提升幅度。企业可仔细列出问题的范围和解决方案带来的预期收益。如果该范围太小，业务收益将无法实现；如果范围太大，获得资金和在恰当的期限内完成项目就会很有挑战性。

对于成本维度，我们还需要考虑是否已有合适的技术人员？大数据解决方案需要特定的技能来理解和分析大数据需求，并维护大数据系统。这些技能包括行业知识、领域专长，以及有关大数据工具和技术的知识。这包括大数据建模、统计、分析等方面的能力。在实施一个新的大数据项目之前，确保已安排了合适的人员，他们熟悉该领域、能分析大量数据，而且能从数据生成有意义且有用的业务机会。

1.7.5　数据治理维度

在决定是否实现一个大数据平台时，企业要特别关注那些新数据源和新的数据元素类型，这些数据所有权可能尚未明确定义。国家的一些规章制度可能会禁止企业获取和使用的数据。例如，在医疗行业，直接获取病人数据是否合法？企业的业务流程可能需要修改，以便能够获取、存储和访问外部数据。下面是一些数据治理的问题。

- 安全性和隐私：在不违反法规和隐私等前提下，可以访问哪些数据？可以存储哪些数据？哪些数据应加密？谁可以查看这些数据？
- 数据的标准化：数据是否有标准格式？是否有专用的格式？部分数据是否为非标准格式？
- 数据可用的时段：数据是否只在一个允许的时段才可用？
- 数据的所有权：谁拥有该数据？是否拥有适当的访问权和权限来使用数据？

- 允许的用法：允许如何使用该数据？

总之，不是所有大数据情形都需要大数据解决方案。竞争对手在做什么？哪些市场力量在发挥作用？客户想要什么？使用上面的几个维度，可以帮助企业确定大数据解决方案是否适合它的业务情形。

1.8 大数据产业链分析

大数据不仅是一个热门词汇，更代表着一个欣欣向荣的产业。2015 年 8 月，国务院常务会议通过了《关于促进大数据发展的行动纲要》，纲要从国家大数据发展战略全局的高度，提出了我国大数据发展的顶层设计，将大数据定为驱动经济增长和社会进步的重要基础国家战略资源。中国的大数据产业是一片广阔的蓝海，从与每个人密不可分的健康医疗到金融个人征信，正融入社会经济发展的方方面面。据分析预测，大数据应用将在国内十多个领域有很大的发展，涵盖万亿市场。最精炼的总结正如马云所提出来的——“未来最重要的能源不是石油，而是数据”。并且，大数据产业的核心是推动数据资源共享与开放，单一的公司是很难发展的，因为在数据领域里面单一公司所需要获得的数据是需要大量的公司去提供。通过大数据资源的开放共享，才能更好地推动大众创业、万众创新。此外，作为云计算、大数据基础的数据中心耗能巨大，绿色数据中心技术将成为全球数据产业的生命线。

各地都在布局大数据产业。比如，2014 年武汉市政府出台《武汉市大数据产业发展行动计划（2014—2018 年）》，通过构建“2+7+N”的大数据产业发展格局，以“中国•武汉光谷”为核心，全面推进武汉市大数据产业发展战略，重点发展左岭大数据产业园等多个产业基地，形成丰富的大数据资源聚集地和完善的产业链，建成国内领先、国际知名的大数据产业和数据资源聚集“洼地”。到 2018 年，实现武汉市大数据产业产值规模 2000 亿元，带动相关产业新增销售收入过万亿元，大数据成为武汉市经济社会发展的新引擎。

1.8.1 技术分析

从技术实施的层次上，我们把整个大数据产业链（或大数据市场）分为以下四个层面，如图 1-4 所示。

大数据应用（政府、金融、运营商、互联网等）、大数据交易、大数据运营
大数据分析工具（数据处理、数据挖掘、可视化、模型预测）
基础软件平台（数据采集、内容管理、数据库）
基础设施（计算、存储和网络）

图 1-4　大数据产业链

最底层是同硬件相关的基础设施层。最上层是同行业相关的大数据应用层，它需要行业的专业知识，使用大数据技术来实施。在有些研究机构中，中间的两层被认为是一层。我们认为，基础软件平台完成数据的汇聚，形成企业的大数据管理层（国外也有人把它叫做数据湖，Data Lake）。在实施了这一层之后，企业或政府单位的数据已经在一个统一平台上了，完成了“数据即服务”的基础平台，实现了全域的数据层。打个生活中做饭的比方，我们已经把油盐酱醋、蔬菜肉类鱼类等食材和调料，都放在冰箱里了。延续上面的比方，那么，大数据分析工具就是做饭的菜刀、锅、搅拌器等等工具。工具的好坏，决定了数据处理和挖掘的效率和结果。大数据分析工具市场是一个竞争化的市场，既有一些新创立的小企业的参与，也有一些类似于谷歌、微软、IBM 等的行业龙头的参与。大数据分析工具的需求是否还会一直持续下去，是否会进入一个成熟阶段是值得我们观察的。

从 2015 年的下半年开始，随着数据的积累，大数据的逐渐深入，越来越多的行业客户（如：银行、政府相关职能部门）清楚了大数据的价值，也清楚了大数据适用的边界。我们已经明显感觉市场在逐渐成熟，开始有正规的独立大数据项目开始招标了，这意味着行业客户已经成熟。随着明确稳定的需求的出现，整个大数据的商业模型就越来越清晰，大数据市场进入了一个新的阶段。

1.8.2　角色分析

整个大数据产业链，可以分成这样四种角色：数据提供商，算法提供商，数据优化提供商和应用提供商。

1. 数据提供商

一般都是由于拥有某种入口资源（比如：运营商、电商等），经过了数年，甚至十数年的积累，形成了在某一领域、某一行业独特的数据资源优势。数据提供商可以将数据提供给第三方使用，从而将资源优势转化成实际的收益。由于分工的细化，数据提供商未必自己去做产业链的其他角色。当然，随着数据成本的日益增高，数据将越来越汇聚到几家巨头手里，而形成几家数据寡头为中心，数家各领域各行业垄断企业为补充的格局。

数据提供商领域依旧属于市场初期。目前没有任何一家数据提供商可以提供所有维度的数据，每家都只拥有部分数据。现在最时髦的各家的“用户画像”也只是盲人摸象。距离真相，还

有一定的距离。一些大数据企业寻找定位某一个行业，苦练内功，成为该行业的大数据的应用服务提供商。从而间接地成为数据提供商（因为数据还是属于购买其应用的客户，所以还是替客户操办的数据提供商）。

2. 算法提供商

这些企业虽然没有数据，但具有行业丰富的经验和背景，可以为客户提供很好的算法服务。目前在各个行业都有一些独立的第三方算法服务提供商。算法提供商将会随着行业应用的深化不断地强化自身在行业的优势，对后来者筑起壁垒。而且，随着在行业经验的积累，算法提供商是最容易成为应用服务提供商的，也可能会被应用服务提供商所取代，不会以单独的形式存在。随着行业应用的深入，每个行业逐渐也会形成几家独大的格局。由算法提供商演变的应用提供商势必会给后来的单纯算法提供商造成很大的壁垒。所以，单纯的算法提供商在未来几年内可能会逐渐淡出。

3. 数据优化提供商

这一角色也没有数据，它需要从数据提供方买来数据（或者由需求方提供数据），然后按照需求方的要求，将数据整理、优化，交付给甲方。至于甲方如何来使用，它并不介入。数据优化提供商既没有足够的数据资源，又没有算法提供商强大的算法和行业洞察能力，所以只能做些低附加值的技术劳务输出。虽然数据优化服务提供商低端，但在整个的产业链里还不容易被取代。随着产业链的日益成熟、分工的日益细化，数据优化服务提供商可能作为一个环节独立存在，而不是作为数据提供商的一环。这一角色，需要精通各种大数据的模型、算法，也需要了解不同数据的特点，从而可以根据用户的需求，为用户“优化”出符合他们需求的数据。

4. 应用提供商

这一角色又叫解决方案提供商，是离客户最近的一个环节，也是最能体现价值的一个环节。对客户而言，他并不关心大数据到底有多大，数据是否足够优化，算法是否足够科学。他关心的是，能否为他解决实际的问题。从这一点上来看，应用提供商颇似一个系统集成商。它需要根据用户的实际需求，去判断需要准备什么样的数据，需要采用什么样的算法，需要将数据如何优化，以便达到最优的效果，帮助客户解决什么样的实际问题。应用提供商需要清楚地知道哪些是大数据能做到的，哪些是大数据做不到的。大数据不是万能的，他需要懂得约束客户的需求和预期。

按照上节中的技术分析，应用提供商可按照技术层次细分为多个角色。应用提供商是大数据市场最关键的角色。数据终究是原材料，能否做出一桌好菜，还要看厨师的手艺。对行业的洞察力和经验，就是对火候的掌握，就是厨师的手艺。

大数据不但有用，而且确确实实可以赚钱。数据作为未来企业的战略资源，的确有着毋庸置疑的重要性，但不至于没有数据，就寸步难行，还没到那种“得数据者得天下”的地步。

上面将整个大数据产业链划分成了四种角色。要想在大数据市场上立足，需要先明白自己属于哪个角色。清楚了自己的身份，清楚自己在产业链的位置。继而沿着自己既定的发展方向坚定

不移地走下去。接下来要做的就是积累，不断地积累和优化，不断地进步，争取做到各自领域的领头羊。想在大数据市场上谋有一席之地，最终靠的还是实力。

今天大数据公司应该做的就是两件事：数据和能力。对于一个大数据公司，你要么有数据，要么有管理和处理数据的能力。没有数据这个生产材料，肯定无法做大数据运营；如果没有足够的驾驭数据的能力，做不出客户满意的效果，也终将会被市场所淘汰。2016 年，将会是大数据领域重要的一年，很有可能将迎来多年我们企盼的拐点的到来。而 2016 年也将是各路诸侯确立自己江湖地位的一年，可以预见，2016 之后，再无大的格局之战！

1.8.3　大数据运营

随着大数据概念的提出，大数据运营公司也犹如雨后春笋般出现。大数据运营貌似很容易，我们只需要：

（1）一套大数据工具
（2）一个或多个分析算法
（3）海量的数据

那么，对于大数据运营公司，该如何找到数据呢？通常有以下几个方面：

- 相信开放数据

不论现在或是将来，这些企业相信开放数据，等待着政府开放更多的数据。这个方法的主要缺点是数据的开放范围往往有限。

- 自己寻找数据

通过网络爬虫等工具来捕获数据。这个方法的主要缺点是，也许那些客户真正感兴趣的数据并不在这个网上。

- 在大企业中寻找数据

首先，他提出一个问题的可能的解决方案（例如，减少欺诈行为，提高你的广告购买的投入，增加你的邮件营销途径，采购性价比更好的原材料，等等）；接着，他说服一个企业来提供数据，并为客户实际解决一个有价值的大数据问题。

- 购买数据

比如：通过大数据交易所购买数据。

1.9 大数据交易

大数据是国家基础性战略资源，大数据发展已成为国家战略，呈现良好的发展趋势。研究资料显示，2011 年至 2015 年全球大数据产业年均复合增长率达到 30%，其中中国市场增速高达 50%。预计 2016 年至 2020 年，中国大数据产业仍将保持 30%以上的年均复合增长率，是同期 IT 市场年均复合增长率的两倍（12%）。未来 5~10 年将是大数据发展的时间窗口。

各地的数据资源非常丰富，已经积累并继续产生庞大的数据资源。以上海为例，上海的医联数据共享系统包含了近 40 家市级医院，接入了 200 多个系统，有 4000 余万的患者健康档案和电子病历。上海还有 4800 万张交通卡、每天 30GB 交通流量信息数据。

随着国家“互联网+”的推进，大数据产业的价值正在凸显。2015 年 4 月 14 日，全国首家大数据交易所落户贵阳，此后，不少城市（如：武汉、上海等）设立大数据交易系统。用于交易的数据有很多，包括来自于通信运营商的数据，广告类、能源类的数据，以及市场采集的各种数据。

虽然大数据交易已经呈现良好的发展趋势，但也存在一些主要问题和不足。比如：大数据资源主要集中在政府部门、公用事业单位和国有企业，数据开放程度不够，而市场数据资源的流通性不够，因为缺乏流通规则，无法深入加工。同时，大数据深度应用少。虽然业界普遍认为大数据应用和产业发展潜力巨大，但现阶段金融、商贸、交通、医疗、制造、能源等行业缺乏应用。最后，则是大数据产业集聚度不高、创业氛围不够活跃。当前全国大数据企业大概有 500 家左右，大数据产业发展较好的园区数量甚少，只有武汉东湖高新技术开发区、上海市北高新园区等极少园区开辟了专门园区来孵化大数据产业。此外，产业政策、知识产权、技术标准等方面还存在不少差距。

大数据交易也涉及许多新的理论问题需要解决。正如华东政法大学知识产权学院院长高富平指出，大数据交易中的各种数据需要清晰界定，弄清楚来源及其合法性，是否涉及个人信息、商业秘密、国家秘密等。如果涉及个人隐私信息、商业秘密和国家秘密信息，那么该数据不可交易；如果涉及个人信息，那么需要进行去身份化处理，才能进入交易。这是大数据交易最难的部分，需要进行研究，制定详细的规则才能进行。再比如，数据的使用不具有排他性，如何界定转让人与受让人之间、使用人与其他使用人之间的关系，就需要建立数据交易规则。为了促进大数据产业的发展，迫切需要国家制定相关法律，规范数据采集、流通与应用，保护数据产权、安全和隐私。很多有识之士就提出了“关于制定大数据法的议案 ”。他们认为：开发数据下的数据交易在法律不完善的情况下将带来许多问题。其中包括交易的模式、交易的公平性、透明性、交易的技术保障、交易的法律法规、知识产权等等。在上述内容没有清晰之前，数据交易一定会出现“数据买卖”乱象。

1.10 大数据之我见

从技术层面看，大数据分析 = 数据 + 算法。如果算法有了，就缺数据。数据如果是分散的，需要数据整合平台；数据可能还是别人的，需要数据交易所。算法怎么来？一个是自己的行业积累，一个是业界的公认算法。一些行业，比如，金融行业，业界已经出了不少的算法，不需要我们自己折腾了。另外一些行业，算法可能还在摸索阶段。

数据整合平台怎么做？本书的前面 9 章都是在讲数据整合的各个步骤，以及相关的产品和技术。第 10 章重点讲了已经有的算法和它们的用法。弄清楚了这些，技术上的大数据分析就清楚了。归根到底，就是怎么弄数据了，到时候才真正地到了“数据为王”的时代了。

第 2 章 大数据软件框架

这是一个大数据时代，而 Hadoop 就是为了大数据应运而生。Hadoop 是 Apache 的子项目，是一个分布式系统基础架构，它主要是用于大数据的处理。比如：如果你要在一个 50TB 的巨型文件中查找内容，会出现什么情况？在传统的系统上，这将需要很长的时间，但是 Hadoop 在设计时就考虑到这些问题，采用分布式存储和并行执行机制。Hadoop 所提供的分布式文件系统（HDFS）实现了大规模的存储（在所有计算节点上分布式存储 50TB 数据），这为整个集群带来了非常高的带宽，因此能大大提高效率。Hadoop 可以让用户在不了解分布式底层细节的情况下，开发分布式程序，充分利用集群的威力进行高速运算和存储。对于大数据而言，Hadoop 就是用大量的廉价机器组成的集群去执行大规模运算，这包括大规模的计算和大规模的存储。

2.1 Hadoop 框架

近年来，Hadoop 已经逐渐成为大数据分析领域最受欢迎的解决方案，像 eBay 这样大型的电子商务企业，一直在使用 Hadoop 技术从数据中挖掘价值，例如，通过大数据提高用户的搜索体验，识别和优化精准广告投放，以及通过点击率分析以理解用户如何使用它的在线市场平台等。目前，eBay 的 Hadoop 集群总节点数超过 10000 多个，存储容量超过 170PB。

Hadoop 框架是用 Java 编写的，它的核心是 HDFS（Hadoop 分布式文件系统）和 MapReduce。HDFS 为大数据提供了存储，MapReduce 为大数据提供了计算。HDFS 可以保存比一个机器的可用存储空间更大的文件，这是因为 HDFS 是一套具备可扩展能力的存储平台，能够将数据分发至成千上万个分布式节点及低成本服务器之上，并让这些硬件设备以并行方式共同处理同一任务。Hadoop 框架实现了名为 MapReduce 的编程范式，这个范式实现了大规模的计算：应用程序被分割成许多小部分，而每个部分在集群中的节点上并行执行（每个节点处理自己的数据）。MapReduce 和分布式文件系统的设计，使得应用程序能够在成千上万的独立计算的电脑上运行并操纵 PB 级的数据。

Hadoop 框架包括 Hadoop 内核、MapReduce、HDFS 和 Hadoop YARN 等。Hadoop 也是一个生态系统，在这里面有很多的组件。除了 HDFS 和 MapReduce，有 NoSQL 数据库的 HBase，有

数据仓库工具 Hive，有 Pig 工作流语言，有机器学习算法库 Mahout，在分布式系统中扮演重要角色的 Zookeeper，有内存计算框架的 Spark，有数据采集的 Flume 和 Kafka。总之，用户可以在 Hadoop 平台上开发和部署任何大数据应用程序。

2.1.1　HDFS（分布式文件系统）

HDFS 是 Hadoop Distribute File System（Hadoop 分布式文件系统）的简称，是 Hadoop 的一个分布式文件系统。HDFS 是一个可运行在廉价机器上的可容错分布式文件系统。它既有分布式文件系统的共同点，又有自己的一些明显的特征。在海量数据的处理中，我们经常碰到一些大文件（几百 GB 甚至 TB 级别）。在常规的系统上，这些大文件的读和写需要花费大量的时间。HDFS 优化了大文件的流式读取方式，它把一个大文件分割成一个或者多个数据块（默认的大小为 64MB），分发到集群的节点上，从而实现了高吞吐量的数据访问，这个集群拥有数百个节点，并支持千万级别的文件。因此，HDFS 非常适合大规模数据集上的应用。

HDFS 设计者认为硬件故障是经常发生的，所以采用了块复制的概念，让数据在集群的节点间进行复制（HDFS 有一个复制因子参数，默认为 3），从而实现了一个高度容错性的系统。当硬件出现故障（如：硬盘坏了）的时候，复制的数据就可以保证数据的高可用性。正是因为这个容错的特点，HDFS 适合部署在廉价的机器上。当然，一块数据和它的备份不能放在同一个机器上，否则这台机器挂了，备份也同样没办法找到。HDFS 使用一种机架位感知的办法，先把一份拷贝放入同机架上的机器，然后再拷贝一份到其他服务器，这台服务器也许是位于不同数据中心的，这样，如果某个数据点坏了，就从另一个机架上调用。除了机架位感知的办法，现在还有基于 erasure code（一种编码存储技术）的方法。这种方法本来是用于通信容错领域的办法，可以节约空间又达到容错的目的，感兴趣的读者可以去查询相关材料。

HDFS 是一个主从结构。如图 2-1 所示，一个 HDFS 集群是由一个名字节点（NameNode）和多个数据节点（DataNode）组成，它们通常配置在不同的机器上。HDFS 将一个文件分割成一个或多个块，这些块被存储在一组数据节点中。名字节点用来操作文件命名空间的文件或目录操作，如：打开、关闭、重命名等等，它同时确定块与数据节点的映射。数据节点负责来自文件系统客户的读写请求。数据节点同时还要执行块的创建、删除，以及来自名字节点的块复制指令。

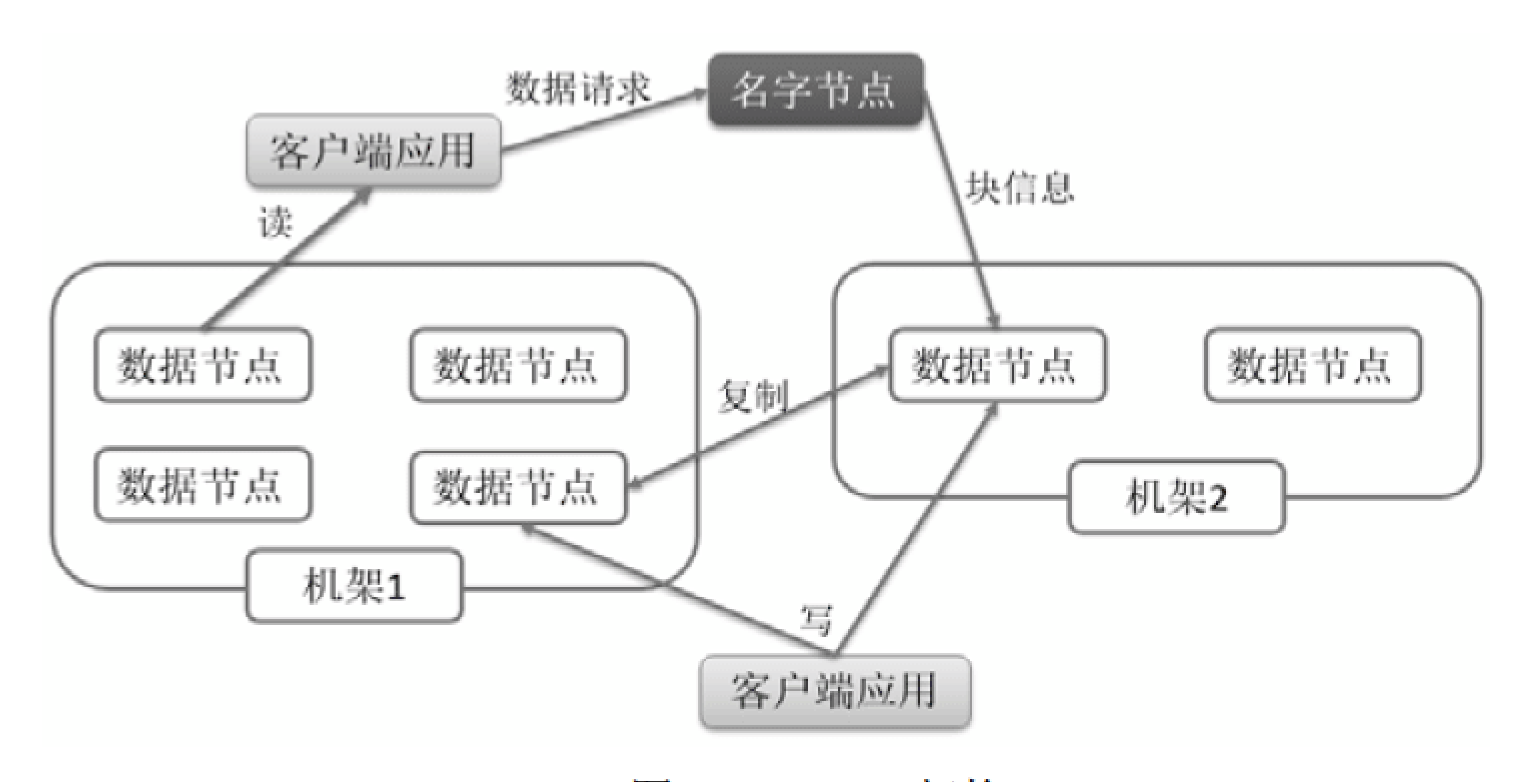


图 2-1　HDFS 架构

一个名字节点保存着集群上所有文件的目录树，以及每个文件数据块的位置信息。它是一个管理文件命名空间和客户端访问文件的主服务器，但是它不真正存储文件数据本身。数据节点通常是一个节点或一个机器，它用来真正地存放文件数据（和复制数据），管理着从 NameNode 分配过来的数据块，并管理对应节点的数据存储。HDFS 对外开放文件命名空间并允许用户数据以文件形式存储。图 2-1 所示显示了一个 HDFS 的架构。

- 客户端应用：每当需要定位一个文件或添加/复制/移动/删除一个文件时，与名字节点交互，获取文件位置信息（返回相关的数据节点信息）；与数据节点交互，读取和写入数据。
- 名字节点（NameNode）：HDFS 文件系统的核心节点，保存着集群中所有数据块位置的一个目录。它管理 HDFS 的名称空间和数据块映射信息，配置副本策略，处理客户端请求。
- 数据节点（DataNode）：存储实际的数据，汇报存储信息给 NameNode。启动后，DataNode 连接到 NameNode，响应 NameNode 的文件操作请求。一旦 NameNode 提供了文件数据的位置信息，客户端应用可以直接与 DataNode 联系。DataNode 并不能感知集群中其他 DataNode 的存在。对于 MapReduce 而言，我们建议把 TaskTracker 实例与 DataNode 部署在同一个服务器上，从而保证 TaskTracker 能够就近访问数据。DataNode 之间可以直接通信，数据复制就是在 DataNode 之间完成的。

名字节点和数据节点都是运行在普通的机器之上的软件，一般都用 Linux 操作系统。因为 HDFS 是用 Java 编写的，任何支持 Java 的机器都可以运行名字节点或数据节点。我们很容易将 HDFS 部署到大范围的机器上。典型的部署是由一个专门的机器来运行名字节点软件，集群中的其他机器每台运行一个数据节点实例。体系结构不排斥在一个机器上运行多个数据节点的实例，但是实际的部署中不会有这种情况。

集群中只有一个名字节点极大地简单化了系统的体系结构。名字节点是仲裁者和所有 HDFS 元数据的仓库，用户的实际数据不经过名字节点。在集群中，我们一般还会配置 Secondary NameNode。这个 Secondary NameNode 下载 NameNode 的 image 文件和 editlogs，并对他们做本地归并，最后再将归并完的 image 文件发回给 NameNode。Secondary NameNode 并不是 NameNode 的热备份，在 NameNode 出故障时并不能工作。

2.1.2 MapReduce（分布式计算框架）

MapReduce 是一种编程模型（也称为计算模型），用以大数据量地批处理计算。如图 2-2 所示，MapReduce 的思想是将批量处理的任务主要分成两个阶段（Map 和 Reduce 阶段），所谓的 Map 阶段就是把数据生成“键-值”对，按键排序。中间有一步叫 shuffle，把同样的 key 运输到同一个 reducer 上面去。在 reducer 上，因为都是同一个 key，就直接可以做聚合（算出总和），最后把结果输出到 HDFS 上。对于应用开发者来说，你需要做的就是编写 Map 和 Reduce 函数，像中间的排序、shuffle 网络传输、容错处理等，框架已经帮你做好了。

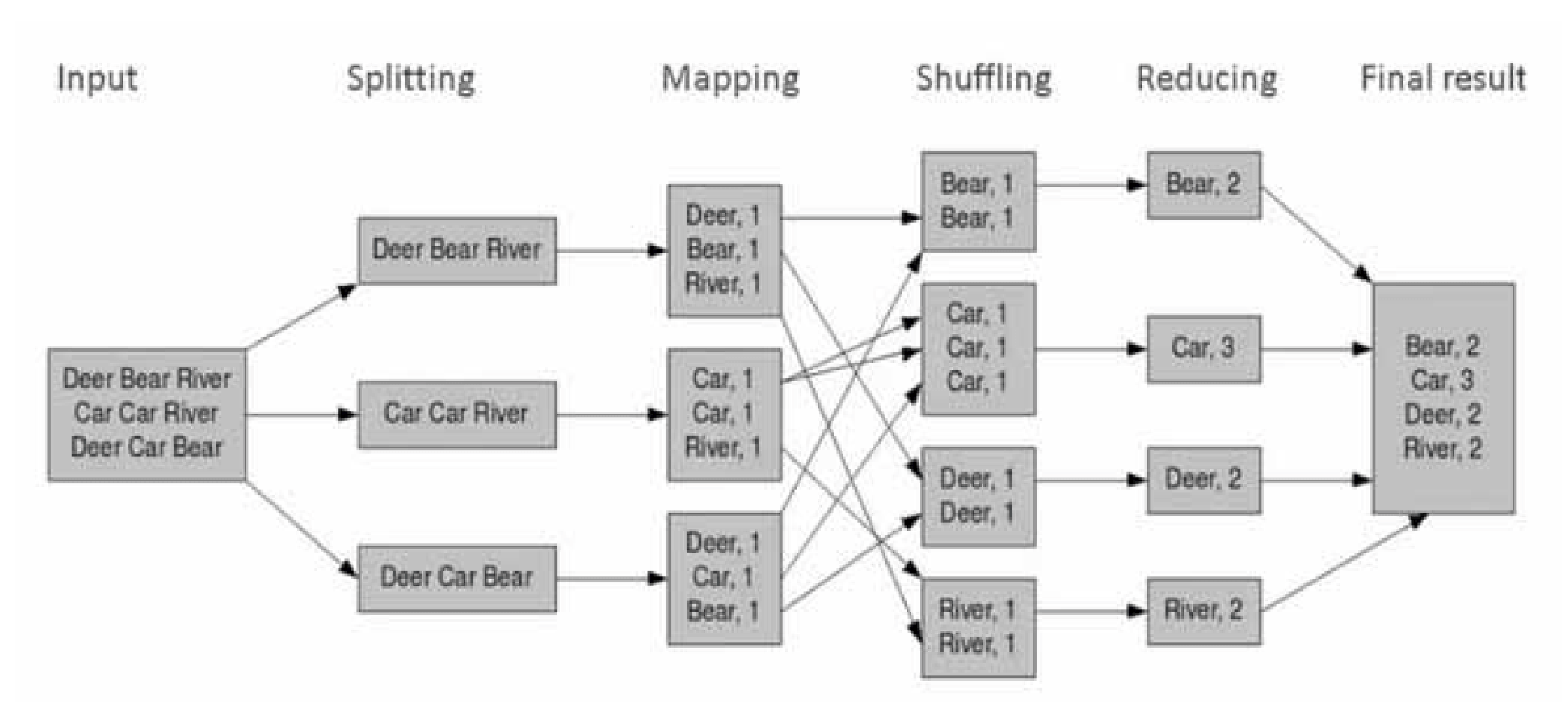


图 2-2　MapReduce 处理示例

MapReduce 的思想和人口普查的做法类似。人口普查委员会给每个城市分配人口普查工作人员（Map 任务），所有人员并行地统计当地的人口数据，最后各个人员的统计数据归约（reduce 任务）到总的人口普查数字。图 2-2 的例子是计算各个单词出现的次数。MapReduce 通常将输入的数据集分割为一些独立的数据块（splitting 步骤），然后由一些 Map 任务（task）在服务器集群上以完全并行的方式进行处理，这些 Map 任务的计算结果最后通过 Reduce 任务合并在一起来计算最终的结果。具体来说，Map 对数据进行指定的操作，生成“键-值”对形式的中间结果（Mapping 步骤，“Deer，1”就是一个键值对，这个中间结果一般存放在文件系统上）。MapReduce 框架对中间结果按照键值排序（Shuffling 步骤），Reduce 则对中间结果中相同“键”的所有“值”进行规约（Reducing 步骤），以得到最终结果。最终结果一般也存放在文件系统上（如 Final result 步骤）。MapReduce 框架负责任务的调度和监控，并重新执行这些失败的任务。MapReduce 非常适合在大量计算机组成的分布式并行环境里进行数据处理。

图 2-3 显示了一个 MapReduce 的使用场景，图中各主要要素说明如下。

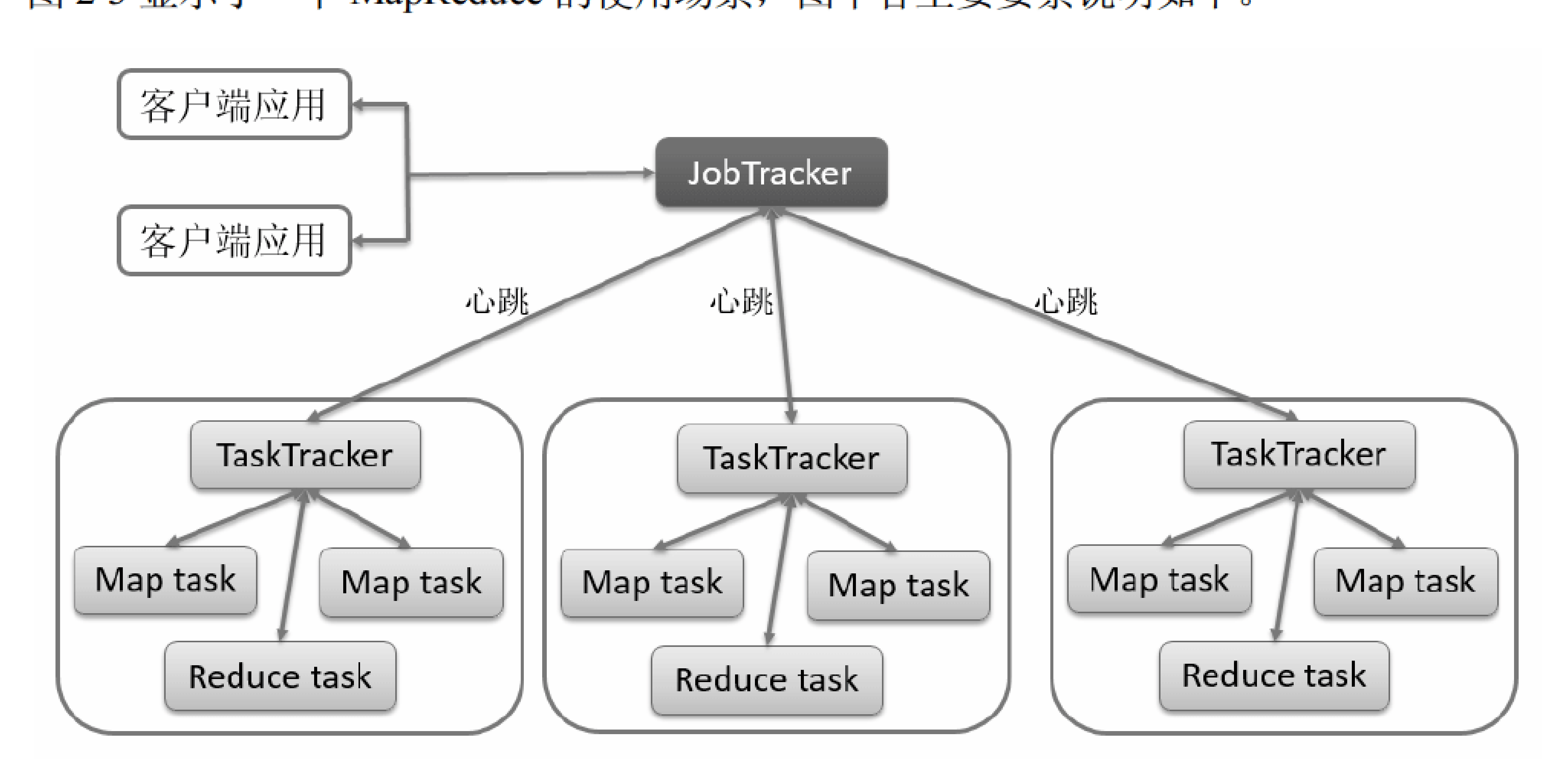


图 2-3　MapReduce 进程示例

- JobTracker：这是主节点，只有一个，它管理所有作业，作业/任务的监控、错误处理等。它将任务分解成一系列的子任务（Map 任务、Reduce 任务、Shuffle 操作），并分派给 TaskTracker。
- TaskTracker：这是从节点，可以有多个，它们接收来自 JobTracker 的 Map Task、Reduce Task 和 Shuffle operations，并执行之。它们与 JobTracker 交互，汇报任务状态。
- Map Task：解析每条数据记录，传递给用户编写的 map()，并执行，最后将输出结果写入本地磁盘（如果为 map-only 作业，直接写入 HDFS）。
- Reduce Task：从 Map Task 的执行结果中，对数据进行排序，将数据按照分组传递给用户编写的 Reduce 函数执行。

我们以一个实际的案例来说明 MapReduce 处理流程。如果我们要统计一下过去 60 年人民日报出现最多的几个词（未必是一个，可能有好几个词出现的次数一样多），那怎么使用 MapReduce 来处理呢？我们首先想到的是，可以写一个程序，把所有人民日报按顺序遍历一遍，统计每一个遇到的词的出现次数，最后就可以知道哪几个词最热门了。但是因为人民日报的文章数量很大，这个方法肯定耗时不少。既然 MapReduce 的本质就是把作业交给多个计算机去完成。那么，我们可以使用上述的程序，部署到 N 台机器上去，然后把 60 年的报纸分成 N 份，一台机器跑一个作业，然后把 N 个运行结果进行整合。MapReduce 本质上就是如此，但是如何拆分 60 年的报纸文件，如何部署程序到 N 个机器上，如何整合结果，这都是 MapReduce 框架定义好的。我们只要定义好这个 map 和 reduce 任务，其他都交给 MapReduce。

下面我们使用 MapReduce 伪代码来说明如何实现 Map 和 Reduce 两个函数。Map 函数和 Reduce 函数是需要我们自己实现的，这两个函数定义了任务本身。MapReduce 计算框架中的输入和输出的基本数据结构是“键-值”对。需要提醒读者的是，我们现在很少直接使用 MapReduce 框架来编写程序了，而是使用基于 MapReduce 框架的工具来编写，或者直接用 Spark 等工具来编写。下面的 Map 函数和 Reduce 函数的伪代码用来帮助读者理解整个 MapReduce 框架的思想。

1. Map 函数

接受一个“键-值”对，产生一组中间“键-值”对。MapReduce 框架有 sort（排序）和 shuffle（发送）操作，sort 会按照键来对 map 函数所产生的键-值对进行排序（如图 2-2 所示的排序结果），然后 shuffle 将所有具有相同键的“键-值”对发送给同一个 reduce 函数。

```
ClassMapper
method map(String input_key, String input_value):
// input_key: 报纸文件名称
// input_value: 报纸文件内容
for each word w in input_value:
   EmitIntermediate(w, "1"); //记录每个词的出现（次数为1）
```

在上面的代码中，map 函数接受的键是文件名（假定文件名是日期，如：20160601 则表明

是 2016 年 6 月 1 日的人民日报电子版文件），值是文件的内容，map 函数逐个遍历词语，每遇到一个词 w，就产生一个中间“键-值”对<w,"1">，这表示这个 w 词出现了一次。

2. Reduce 函数

接受一个键（一个词），以及相关的一组值（这一组值是所有 Map 对于这个词计算出来的频数的一个集合），整个输入数据也是一个“键-值”对。将这组值进行合并产生一组规模更小的值（通常只有一个或零个值）。在我们这个例子中，是将值集合中的频数进行求和，然后记录每个词和这个词出现的总频数。

```
ClassReducer
method reduce(String output_key,Iterator intermediate_values):
// output_key: 一个词
// intermediate_values: 该词对应的所有出现次数的列表
int result = 0;
for each v in intermediate_values:
result += ParseInt(v); //次数累加
Emit(AsString(result));//记录最终累加结果
```

MapReduce 将键相同（都是词 w）的“键-值”对传给 reduce 函数，这样 reduce 函数接受的键就是单词 w，值是字符串“1”的列表（键为 w 的键-值对的个数），然后将这些“1”累加就得到单词 w 的出现次数。最后存储在 HDFS 上。

MapReduce 支持 C/C++、Java、Ruby、Perl 和 Python 编程语言。开发人员可以使用 MapReduce 库来创建任务。至于节点之间的通信和协调，输入数据集的切割，在不同机器之间的程序执行调度，处理错误等，这些都由框架完成，开发人员无须处理。Map 和 Reduce 函数会自动在多个服务器节点上自动并行执行。即使开发人员完全没有并行和分布式系统的经验和知识，也能轻松地利用好大型分布式系统的资源。MapReduce 革新了海量数据计算的方式，为运行在成百上千台机器上的并行程序提供了简单的编程模型。MapReduce 几乎可以做到线性扩展：随着数据量的增加，可以通过增加更多的计算机来保持作业时间不变。MapReduce 容错性强，它将工作拆分成多个小任务后，能很好地处理任务失败。

2.1.3　YARN（集群资源管理器）

从 Hadoop 2 开始，MapReduce 被一个改进的版本所替代，这个新版本叫做 MapReduce 2.0（MRv2）或 YARN（Yet Another Resouce Negotiator，另一种资源协调者）。YARN 是一种新的 Hadoop 资源管理器，也是一个通用资源管理系统，可为上层应用提供统一的资源管理和调度，它的引入为集群在利用率、资源统一管理和数据共享等方面带来了巨大好处。我们先来回顾一下老版本的 MapReduce 的流程和设计思路。从图 2-3 可以看出：

- 首先客户端应用提交了一个 job，job 的信息会发送到 Job Tracker 中，Job Tracker 是 MapReduce 框架的中心，它与集群中的机器定时通信（心跳），确定哪些程序在哪些机

器上执行，管理所有 job 失败、重启等操作。

- TaskTracker 在 MapReduce 集群中每台机器都有，主要是监视所在机器的资源情况。
- TaskTracker 同时监视当前机器上的 tasks 运行状况。TaskTracker 把这些信息通过心跳发送给 JobTracker，JobTracker 会搜集这些信息，为新提交的 job 确定运行在哪些机器上。

MapReduce 架构简单明了，在最初推出的几年，得到了众多的成功案例，获得业界广泛的支持和肯定。但随着分布式系统集群的规模及其工作负荷的增长，原框架固有的问题逐渐浮出水面，主要的问题集中如下：

- JobTracker 是 MapReduce 的集中处理点，存在单点故障。
- JobTracker 承担了太多的任务，造成了过多的资源消耗，当 job 非常多的时候，会造成很大的内存开销，也增加了 JobTracker 崩溃的风险。业界的共识是老版本的 MapReduce 的上限只能支持 4000 个节点主机。
- 在 TaskTracker 端，只以 map/reduce task 的数目作为资源的表示过于简单，没有考虑到 CPU 和内存的占用情况，如果两个大内存消耗的 task 被调度到了一块，很容易出现 Java 的 OOM。
- 在 TaskTracker 端，把资源强制划分为 map task slot 和 reduce task slot。当系统中只有 map task 或者只有 reduce task 的时候，这会造成资源的浪费，也就是前面提到的集群资源利用的问题。

YARN 最初是为了修复 MapReduce 实现里的明显不足，并对可伸缩性（支持一万个节点和二十万个内核的集群）、可靠性和集群利用率进行了提升。YARN 把 Job Tracker 的两个主要功能（资源管理和作业调度/监控）分成了两个独立的服务程序——全局的资源管理（Resource Manager，简称为 RM）和针对每个应用的 App Master（AM），这里说的应用要么是传统意义上的 MapReduce 任务，要么是任务的有向无环图（DAG）。Resource Manager 和每一台机器的节点管理服务器（Node Manager）能够管理用户在哪台机器上的进程，并能对计算进行组织。其架构图如图 2-4 所示。

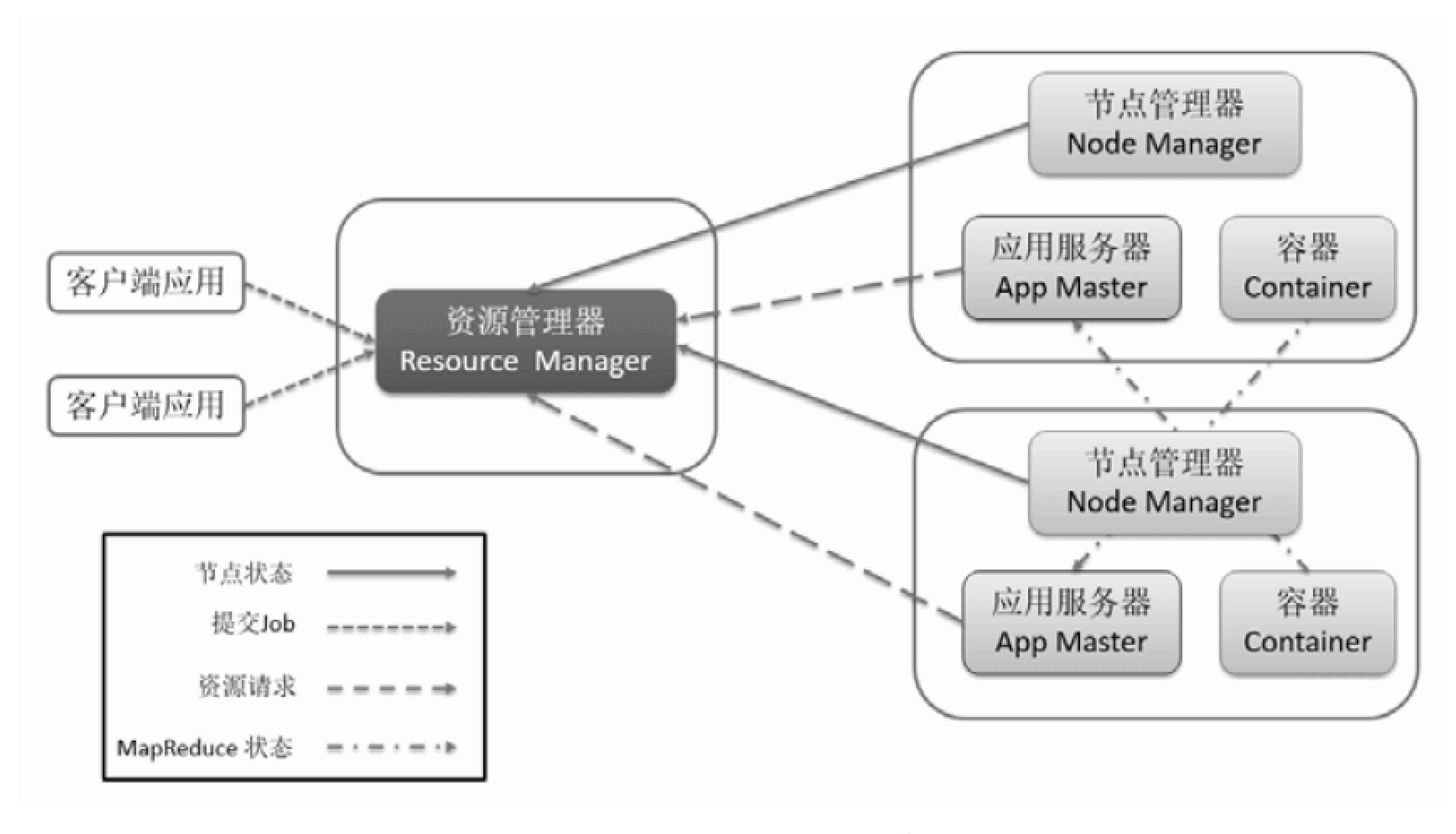


图 2-4　YARN 架构

Resource Manager 支持分层级的应用队列，这些队列享有集群一定比例的资源。它是一个调度器，可以基于应用程序对资源的需求进行调度。每一个应用程序需要不同类型的资源，因此就需要不同的容器（container）。资源包括：内存、CPU、磁盘、网络等等。可以看出，这同老版本 MapReduce 的固定类型的资源使用模型有显著区别。资源管理器提供一个调度策略的插件，它负责将集群资源分配给多个队列和应用程序。

图中 Node Manager 是每一台机器的代理，是执行应用程序的容器，它监控应用程序的资源使用情况（CPU、内存、硬盘、网络）并且向资源管理器汇报。每一个应用的 Application Master 是一个框架库，它结合从 Resource Manager 获得的资源和 Node Manager 协同工作来运行和监控任务。每一个应用的 Application Master 向资源管理器索要适当的资源容器，运行任务，跟踪应用程序的状态和监控它们的进程，处理任务的失败。

让我们对新旧 MapReduce 框架做一下比较。首先客户端应用不变，其 API 大部分保持兼容，这也是为了对开发者透明化，使其不必对原有应用代码做大的修改，但是原框架中核心的 JobTracker 和 TaskTracker 不见了，取而代之的是 Resource Manager、Application Master 与 NodeManager 三个部分。Resource Manager 是一个中心的服务，它是调度和启动每一个 Job 所属的 Application Master，另外监控 Application Master 的存在情况。在老版本的 Job 里面所在的 task 监控和重启在 YARN 中都不见了，这就是在 YARN 中出现 Application Master 的原因。Resource Manager 负责作业与资源的调度，接收 Job Submitter 提交的作业，按照作业的上下文（Context）信息，以及从 Node Manager 收集来的状态信息，启动调度过程，分配一个 Container。Node Manager 功能比较专一，就是负责 Container 状态的维护，并向 Resource Manager 保持心跳。Application Master 负责一个 Job 生命周期内的所有工作，类似老框架中的 JobTracker。但需要注意每一个 Job（不是每一种）都有一个 Application Master，它可以运行在 Resource Manager 以外的机器上。

Yarn 框架相对于老的 MapReduce 框架有什么优势呢？首先，这个设计大大减小了 JobTracker（也就是 YARN 的 Resource Manager）的资源消耗，并且让监测每一个 Job 子任务（tasks）状态的程序分布式了。对于资源的表示以内存为单位，比之前以剩余 slot 数目为单位更合理。老的框架中，JobTracker 一个很大的负担就是监控 job 下 tasks 的运行状况，现在，这个部分由 Application Master 完成，而 Resource Manager 中有一个模块叫做 ApplicationsMasters（注意不是 ApplicationMaster），它用来监测 ApplicationMaster 的运行状况，如果出问题，会将其在其他机器上重启。Container 是 Yarn 为了将来作资源隔离而提出的一个框架。这一点应该是借鉴了 Mesos 的工作机制，虽然 Container 目前是一个框架，仅仅提供 Java 虚拟机内存的隔离，但是未来可能会支持更多的资源调度和控制。既然资源表示成内存量，那就没有了之前的 map slot/reduce slot 分开所造成集群资源闲置的问题。

总之，YARN 从某种意义上来说应该算是一个云操作系统，它负责集群的资源管理。在操作系统之上可以开发各类的应用程序。这些应用可以同时利用 Hadoop 集群的计算能力和丰富的数据存储模型，共享同一个 Hadoop 集群和驻留在集群上的数据。此外，这些新的框架还可以利用 YARN 的资源管理器，提供新的应用管理器实现。本书后面介绍的 Spark 框架就支持 YARN。

2.1.4 Zookeeper（分布式协作服务）

Zookeeper 是一个集中式服务，主要负责分布式任务调度，它用来完成配置管理、名字服务、提供分布式锁以及集群管理等工作。具体说明如下。

1. 配置管理

应用程序中经常有一些配置，比如数据库连接等。一般我们都是使用配置文件的方式，在代码中引入这些配置文件。这种方式适合只有一台服务器的时候。当我们有很多服务器时，就需要寻找一种集中管理配置的方法，而不是在各个服务器上存放配置文件。我们在这个集中的地方修改了配置，所有需要配置的服务都能读取配置。一般我们用一个集群来提供这个配置服务以提升可靠性。

Zookeeper 保证了配置在集群中的一致性，它使用 Zab 这种一致性协议来提供一致性。现在有很多开源项目使用 Zookeeper 来维护配置，比如在 HBase 中，客户端就是连接一个 Zookeeper，获得必要的 HBase 集群的配置信息，然后才可以进一步操作。在开源的消息队列 Kafka 中，也使用 Zookeeper 来维护 broker 的信息。

2. 名字服务

DNS 把域名（如：www.da-shuju.com）对应到 IP 地址（59.175.137.94），从而为我们提供了名字服务。在应用系统中我们有时也会需要这类名字服务，特别是在服务特别多的时候。我们只需要访问一个共同的地方，它提供统一的入口。

3. 分布式锁

Zookeeper 是一个分布式协调服务。我们利用 Zookeeper 来协调多个分布式进程之间的活动。在一个分布式环境中，为了提高可靠性，集群中的每台服务器上都部署着同样的服务。我们使用分布式锁，在某个时刻只让一个服务去干活，当这个服务出问题时就将锁释放，并立即切换到另外的服务上。比如 HBase 的 Master 就是采用这种机制。在 Zookeeper 中是通过选举 leader 完成的分布式锁。

4. 集群管理

在分布式的集群中，经常会由于各种原因，比如硬件故障、软件故障、网络问题，有新的节点加入进来，也有老的节点退出集群。这个时候，集群中其他机器需要感知到这种变化，然后根据这种变化做出对应的决策。比如：一个分布式的 SOA 架构中，服务是一个集群提供的，当消费者访问某个服务时，就需要确定哪些节点可以提供该服务。Kafka 的队列就采用了 Zookeeper 作为消费者的上下线管理。

总之，Zookeeper 就是一种可靠的、可扩展的、分布式的、可配置的协调机制，用来统一分布式系统的状态。

2.1.5 Ambari（管理工具）

Apache Ambari 是一种基于 Web 的 Hadoop 管理工具，可以快捷地监控、部署、管理 Hadoop 集群。Ambari 目前已支持大多数 Hadoop 组件，包括 HDFS、MapReduce、Hive、Pig、HBase、Zookeper、Sqoop 和 Hcatalog 等。Ambari 可以帮助 Hadoop 系统管理员来完成以下工作：

- 通过一步一步的安装向导简化了集群的安装和配置。
- 集中管理（包括：启动、停止和重新配置）集群上的 Hadoop 服务。
- 预先配置好关键的运维指标，可以直接查看 Hadoop Core（HDFS 和 MapReduce）及相关项目（如 HBase、Hive 和 HCatalog）是否健康。
- 支持作业与任务执行的可视化与分析，能够更好地查看依赖和性能。
- 通过一个完整的 RESTful API 把监控和管理功能嵌入到自己的应用系统中。
- 用户界面非常直观，用户可以轻松有效地查看信息并控制集群。

Ambari 使用 Ganglia 收集度量指标，用 Nagios 支持系统报警，当需要引起管理员的关注时（比如，节点停机或磁盘剩余空间不足等问题），系统将向其发送邮件。此外，Ambari 能够安装安全的（基于 Kerberos）Hadoop 集群，以此实现了对 Hadoop 安全的支持，提供了基于角色的用户认证、授权和审计功能，并为用户管理集成了 LDAP 和 Active Directory。

2.2 Spark（内存计算框架）

随着大数据的发展，人们对大数据的处理要求也越来越高，原有的批处理框架 MapReduce 适合离线计算，却无法满足实时性要求较高的业务，如实时推荐、用户行为分析等。因此，Hadoop 生态系统又发展出以 Spark 为代表的新计算框架。相比 MapReduce，Spark 速度快，开发简单，并且能够同时兼顾批处理和实时数据分析。

Apache Spark 是加州大学伯克利分校的 AMPLabs 开发的开源分布式轻量级通用计算框架，并于 2014 年 2 月成为 Apache 的顶级项目。由于 Spark 基于内存设计，使得它拥有比 Hadoop 更高的性能，并且对多语言（Scala、Java、Python）提供支持。Spark 有点类似 Hadoop MapReduce 框架。Spark 拥有 Hadoop MapReduce 所具有的优点；但不同于 MapReduce 的是 Job 中间输出结果可以保存在内存中，从而不再需要读写 HDFS（MapReduce 的中间结果要放在文件系统上），因此，在性能上，Spark 能比 MapReduce 框架快 100 倍左右（如图 2-5 所示），排序 100TB 的数据只需要 20 分钟左右。正是因为 Spark 主要是在内存中执行，所以 Spark 对内存的要求非常高，一个节点通常需要配置 24GB 的内存。在业界，我们有时把 MapReduce 称为批处理计算框架，把 Spark 称为实时计算框架、内存计算框架或流式计算框架。

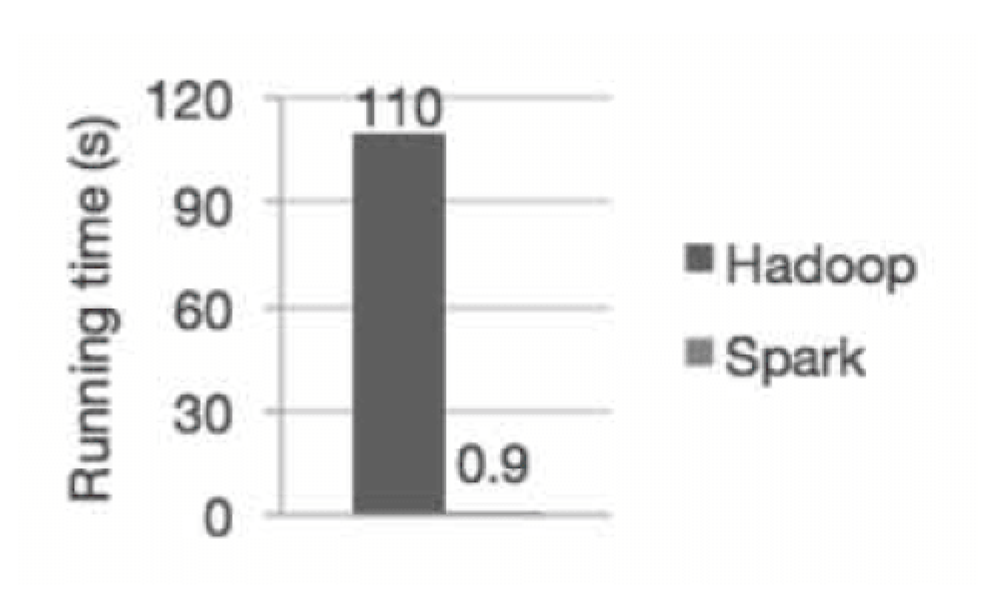


图 2-5　性能比较(数据来源：http://spark.apache.org/)

Hadoop 使用数据复制来实现容错性（I/O 高），而 Spark 使用 RDD（Resilient Distributed Datasets，弹性分布式数据集）数据存储模型来实现数据的容错性。RDD 是只读的、分区记录的集合。如果一个 RDD 的一个分区丢失，RDD 含有如何重建这个分区的相关信息。这就避免了使用数据复制来保证容错性的要求，从而减少了对磁盘的访问。通过 RDD，后续步骤如果需要相同数据集时就不必重新计算或从磁盘加载，这个特性使得 Spark 非常适合流水线式的处理。

虽然 Spark 可以独立于 Hadoop 来运行，但是 Spark 还是需要一个集群管理器和一个分布式存储系统。对于集群管理，Spark 支持 Hadoop YARN、Apache Mesos 和 Spark 原生集群。对于分布式存储，Spark 可以使用 HDFS、Casandra、OpenStack Swift 和 Amazon S3。Spark 支持 Java、Python 和 Scala（Scala 是 Spark 最推荐的编程语言，Spark 和 Scala 能够紧密集成，Scala 程序可以在 Spark 控制台上执行）。应该说，Spark 紧密集成 Hadoop 生态系统中的上述工具。Spark 可以与 Hadoop 上的常用数据格式（如：Avro 和 Parquet）进行交互，能读写 HBase 等 NoSQL 数据库，它的流处理组件 Spark Streaming 能连续从 Flume 和 Kafka 之类的系统上读取数据，它的 SQL 库 Spark SQL 能和 Hive Metastore 交互。

Spark 可用来构建大型的、低延迟的数据分析应用程序。如图 2-6 所示，Spark 包含了如下的库：Spark SQL，Spark Streaming，MLlib（用于机器学习）和 GraphX。其中 Spark SQL 和 Spark Streaming 最受欢迎，大概 60%左右的用户在使用这两个中的一个。而且 Spark 还能替代 MapReduce 成为 Hive 的底层执行引擎。

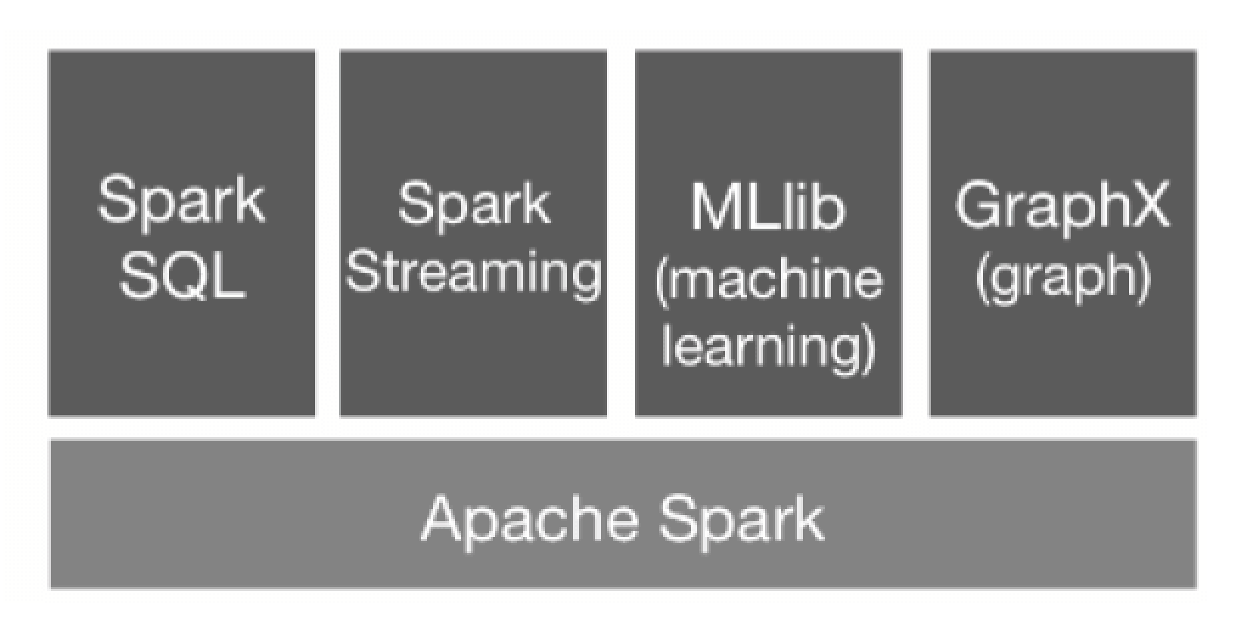


图 2-6　Spark 组件

Spark 的内存缓存使它适合于迭代计算。机器学习算法需要多次遍历训练集，可以将训练集缓存在内存里。在对数据集进行探索时，数据科学家可以在运行查询的时候将数据集放在内存，这样就节省了访问磁盘的开销。

虽然 Spark 目前被广泛认为是下一代 Hadoop，但是 Spark 本身的复杂性也困扰着开发人员。Spark 的批处理能力仍然比不过 MapReduce，Spark SQL 和 Hive 的 SQL 功能相比还有一定的差距，Spark 的统计功能与 R 语言还没有可比性。

2.2.1 Scala

Spark 框架是用 Scala 开发的，并提供了 Scala 语言的一个子集。那么，什么是 Scala 呢？Scala 是一种类似 Java 的编程语言，它的设计初衷是创造一种更好地支持组件的语言。Scala 的编译器把源文件编译成 Java 的 class 文件，从而让 Scala 程序运行在 JVM 上。Scala 兼容现有的 Java 程序，从 Scala 中可调用所有的 Java 类库。Scala 能够让我们花更少的时间和更少的代码编写一样功能的 Java 程序。在 JVM 上，Scala 代码多了一个运行库 scala-library.jar。

Scala 支持交互式运行，开发人员无须编译就能运行这个代码。比如：键入下列 Scala 代码，然后按 Enter 键：

```
scala> println("Hello, Scala!");
```

这将产生以下结果：

```
Hello, Scala!
```

Scala 和 Java 间的最大语法的区别在于“;”（行结束符）是可选的。其他都非常类似。下面我们来编写一个简单的 Scala 代码，用于打印简单的一句话：“Hello, World!”。

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!") // prints Hello World
  }
}
```

其中的“def main(args: Array[String])”是 Scala 程序的 main()，这是每一个 Scala 程序的入口部分。我们将上述代码保存为 HelloWorld.scala 文件，然后输入“scalac HelloWorld.scala”编译该代码。这将在当前目录中生成几个类文件。其中一个名称为 HelloWorld.class。这是一个可以运行在 Java 虚拟机（JVM）上的字节码。键入“scala HelloWorld”来运行程序，就可以在窗口上看到“Hello, World!”。

Spark 框架是用 Scala 语言编写的，在使用 Saprk 时，采用与底层框架相同的编程语言有很多好处：

- 性能开销小；
- 能用上 Spark 最新的版本；
- 有助于你更理解 Spark 的原理。

感兴趣的读者可以参考《Scala 编程思想》一书深入学习 Scala 编程。

2.2.2 Spark SQL

Spark 的存在是为了以快于 MapReduce 的速度进行分布式计算。Spark 的设计者很快就了解到，大家还是想要用 SQL 来访问数据，于是 Spark SQL 就出现了。Spark SQL 是基于 Spark 引擎对 HDFS 上的数据集或已有的 RDD 执行 SQL 查询。有了 Spark SQL 就能在 Spark 程序里用 SQL 语句操作数据了。比如：

```
    val sqlContext = new org.apache.spark.sql.SQLContext(sc)
    val persons = sqlContext.sql("SELECT name FROM people WHERE age >= 18
AND age <= 29")
```

上述两行代码是 Scala 的语法。这两行都声明了两个新变量。与 Java 不同的是，Scala 在变量声明时不给定变量类型。这个功能在 Scala 编程语言中称为类型推断。Scala 会从上下文中分析出变量类型。只要在 Scala 中定义新变量，必须在变量名称之前加上 val 或 var。带有 val 的变量是不可变变量，一旦给不可变变量赋值，就不能改变。而以 var 开头的变量则可以改变值。

Spark SQL 在 Spark 圈中非常流行。Spark SQL 的前身是 Shark。我们简短回顾一下 Shark 的整个发展历史。对于熟悉 RDBMS 但又不理解 MapReduce 的技术人员来说，Hive 提供快速上手的工具，它是第一个运行在 Hadoop 上的 SQL 工具。Hive 基于 MapReduce，但是 MapReduce 的中间过程消耗了大量的 I/O，影响了运行效率。为了提高在 Hadoop 上的 SQL 的效率，一些工具开始产生，其中表现较为突出的是：MapR 的 Drill、Cloudera 的 Impala、Shark。其中 Shark 是伯克利实验室 Spark 生态环境的组件之一，它修改了内存管理、物理计划、执行三个模块，并使之能运行在 Spark 引擎上，从而使得 SQL 查询的速度得到 10~100 倍的提升。Shark 依赖于 Hive，比如：Shark 采用 Hive 的语法解析器和查询优化器，这制约了 Spark 各个组件的相互集成，所以提出了 Spark SQL 项目。2014 年 6 月 1 日，Shark 项目组宣布停止对 Shark 的开发，将所有资源放在 Spark SQL 项目上。Spark SQL 作为 Spark 生态的一员继续发展，而不再受限于 Hive，只是兼容 Hive。Spark SQL 体系架构如图 2-7 所示。

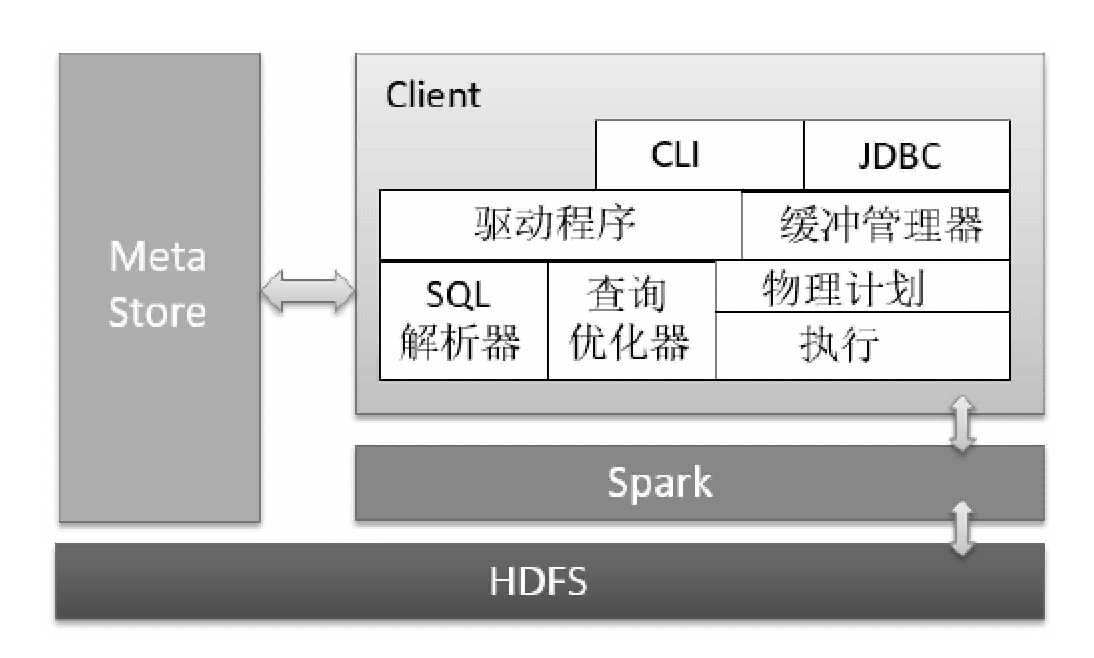


图 2-7　Spark SQL 体系架构

Spark SQL 抛弃原有 Shark 的代码，汲取了 Shark 的一些优点，如内存列存储（In-Memory Columnar Storage）、Hive 兼容性等，重新开发了 Spark SQL 代码。由于摆脱了对 Hive 的依赖性，Spark SQL 无论在数据兼容、性能优化、组件扩展方面都得到了极大的方便。在数据兼容方面，Spark 不但兼容 Hive，还可以从 RDD、parquet 文件、JSON 文件中获取数据，未来版本甚

至支持获取 RDBMS 数据以及 cassandra 等 NOSQL 数据。在性能优化方面，除了采取内存列存储、字节码生成技术（bytecode generation）等优化技术外，将会引进 Cost Model 对查询进行动态评估、获取最佳物理计划等等。在组件扩展方面，无论是 SQL 的语法解析器还是优化器，都可以重新定义并进行扩展。

2.2.3 Spark Streaming

Spark Streaming 是基于 Spark 引擎对数据流进行不间断处理。只要有新的数据出现，Spark Streaming 就能对其进行准实时（数百毫秒级别的延时）的转换和处理。Spark Streaming 的工作原理是在小间隔里对数据进行汇集从而形成小批量，然后在小批量数据上运行作业。

使用 Spark Streaming 编写的程序与编写 Spark 程序非常相似，在 Spark 程序中，主要通过操作 RDD 提供的接口，如 map、reduce、filter 等，实现数据的批处理。而在 Spark Streaming 中，则通过操作 DStream（表示数据流的 RDD 序列）提供的接口，这些接口和 RDD 提供的接口类似。下面我们来看一个应用案例。

假定有一个电商网站，它买了几个搜索引擎（如：百度）的很多关键词。当用户在各大搜索引擎上搜索数据时，搜索引擎会根据购买的关键字导流到电商网站的相关产品页面上，用户可能会购买这些产品。现在需要分析的是哪些搜索词带来的订单比较多，然后根据分析结果多投放这些转化率比较高的关键词，从而为电商网站带来更多的收益。

原先的做法是每天凌晨分析前一天的日志数据，这种方式实时性不高，而且由于日志量比较大，单台机器处理已经达到了瓶颈。现在选择了使用 Spark Streaming + Kafka+Flume 来处理这些日志，并且运行在 YARN 上以应对遇到的问题。

如图 2-8 所示，业务日志分布在各台服务器上。由于业务量比较大，所以日志都是按小时切分的，我们采用 Flume 实时收集这些日志（图中步骤 1），然后发送到 Kafka 集群（图中步骤 2）。这里为什么不直接将原始日志直接发送到 Spark Streaming 呢?这是因为，如果 Spark Streaming 挂掉了，也不会影响到日志的实时收集。

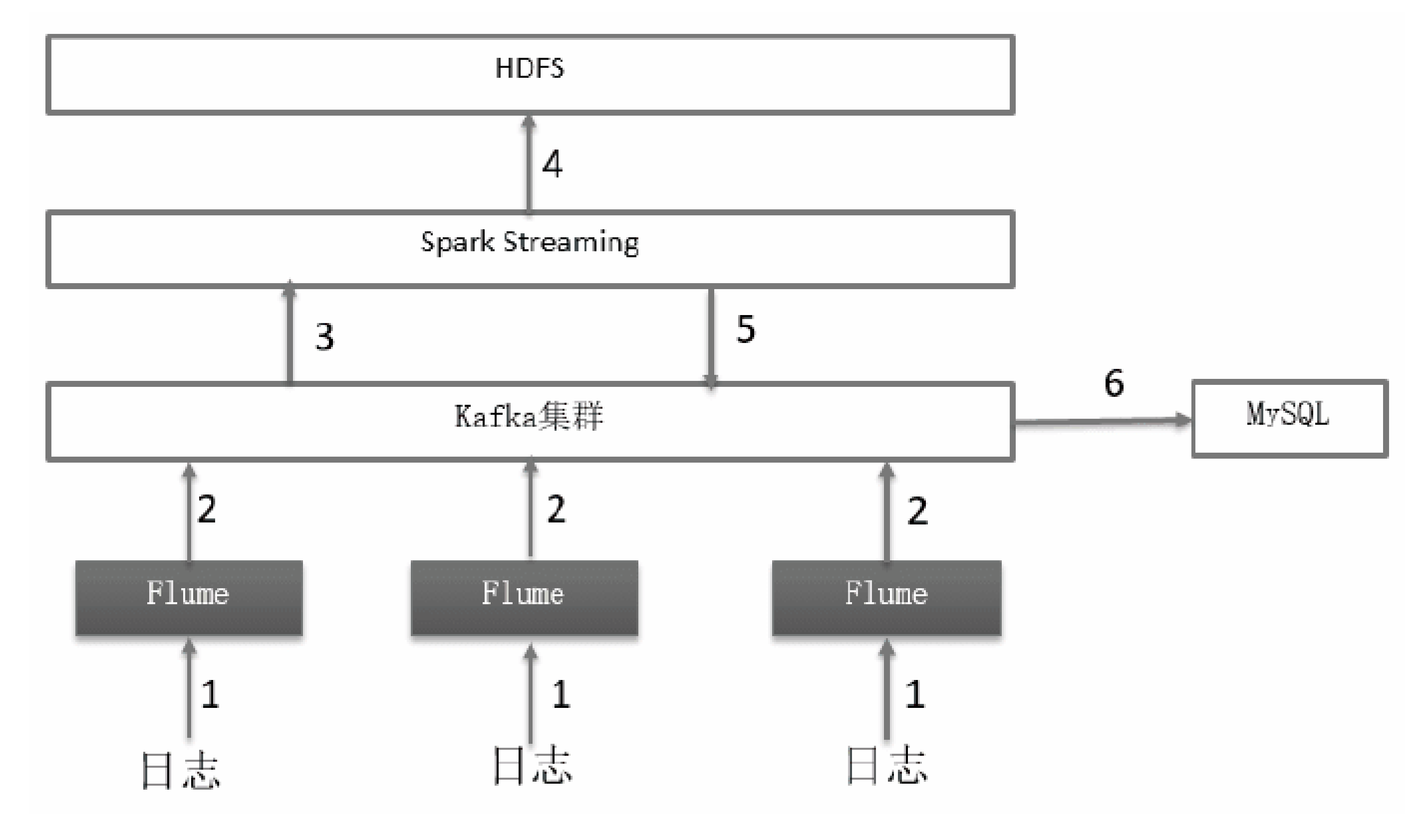


图 2-8 Spark Streaming 应用案例

日志实时到达 Kafka 集群后，我们再通过 Spark Streaming 实时地从 Kafka 拉数据（图中

步骤 3)，然后解析日志，并根据一定的逻辑过滤数据和分析订单和搜索词的关联性。我们使用 Spark 的 KafkaUtils.createDirectStream API 从 Kafka 中拉数据，代码片段如下：

```
    val sparkConf = new SparkConf().setAppName("OrderSpark")
    val sc = new SparkContext(sparkConf)
    val ssc = new StreamingContext(sc, Seconds(2))
    val kafkaParams = Map[String, String]("metadata.broker.list" ->
brokerAddress, "group.id" -> groupId)
    val messages = KafkaUtils.createDirectStream[String, String,
StringDecoder, StringDecoder](ssc, kafkaParams, Set(topic))
```

在上述代码中返回的 messages 是一个刚刚创建 DStream，它是对 RDD 的封装，其上的很多操作都类似于 RDD 。createDirectStream 函数是 Spark 1.3.0 开始引入的，其内部实现是调用 Kafka 的低层次 API，Spark 本身维护 Kafka 偏移量等信息，所以可以保证数据零丢失。

为了能够在 Spark Streaming 程序挂掉后又能从断点处恢复，我们每隔 2 秒进行一次 Checkpoint，这些 Checkpoint 文件存储在 HDFS 上（图中步骤 4）的 Checkpoint 目录中。我们可以在程序里面设置 Checkpoint 目录：

```
    ssc.checkpoint(checkpointDirectory)
```

如果我们需要从 Checkpoint 目录中恢复，我们可以使用 StreamingContext 中的 getOrCreate 函数。为了让分析结果共享给其他系统使用，我们将分析后的数据重新发送到 Kafka（图中步骤 5）。最后，我们单独启动了一个程序从 Kafka 中实时地将分析好的数据存到 MySQL 中用于持久化存储（图中步骤 6）。

2.3 实时流处理框架

在大数据领域，Hadoop 无疑是炙手可热的技术。作为分布式系统架构，Hadoop 具有高可靠性、高扩展性、高效性、高容错性和低成本的优点。然而，随着数据体量越来越大，实时处理能力成为了许多客户需要面对的首要挑战。Hadoop 的 MapReduce 是一个批处理计算框架，在实时计算处理方面显得十分乏力。Hadoop 生态圈终于迎来了实时流处理框架。除了实时性，流处理可以处理更复杂的任务，能够以低延时执行大部分批处理的工作任务。一个典型的流架构如图 2-9 所示，由三个步骤组成：



图 2-9　典型的流架构

- 采集模块组件是从各种数据源收集数据流（如图 2-8 的步骤 1）；
- 集成模块组件集成各种数据流，使它们可用于直接消费（如图 2-8 的步骤 2）；
- 分析模块组件用来分析消费这些流数据。

这三个步骤中，第一步是从各种数据源收集事件，如图 2-8 的 Flume 组件。这些事件来自于数据库、日志、传感器等，这些事件需要清理组织化到一个中心。第二步，在一个中心集成各种流，典型工具如图 2-8 所示的 Apache Kafka。Kafka 提供一个 broker 功能，以高可靠性来收集和缓冲数据，并分发到各种对不同流感兴趣的消费者那里进行分析。第三步，对流进行真正的分析，比如创建计数器实现聚合，Map/Reduce 之类计算，将各种流 Join 一起分析等等，提供了数据分析所需的一步到位的高级编程。

在 Apache 下有多个流处理系统，例如：Apache Kafka、Apache Storm、Apache Spark Streaming、Apache Flink 等。尽管 Spark 比 Hadoop 要快很多，但是 Spark 还不是一个纯流处理引擎。Spark 只是一个处理小部分输入数据的快速批操作（微批处理模式）。这就是 Flink 与 Spark 流处理的区别。Spark 流处理提供了完整的容错功能，并保证了对流数据仅一次处理（也就是说，如果一个节点失败，它能恢复所有的状态和结果）。这是 Flink 和 Storm 所不能提供的功能。Flink 和 Storm 的应用开发人员需要考虑数据丢失的情况，这也限制了开发人员开发复杂的应用系统。

2.4 框架的选择

大数据系统架构有两个组成部分，实时数据流处理和批量数据处理。我们根据具体的需求选择适当的数据处理框架。一些框架适用于批量数据处理，而另外一些适用于实时数据处理。一些框架使用内存模式，另外一些是基于磁盘 I/O 处理模式。基于内存的框架性能明显优于基于磁盘 I/O 的框架，但是同时成本也高很多。总之，要选择一个能够满足需求的框架。否则就有可能既无法满足功能需求，也无法满足非功能需求（比如：性能需求）。

一些框架将数据划分成较小的块。这些小数据块由各个作业独立处理。协调器管理所有这些独立的子作业。数据分块是需要小心的。数据块越小，就会产生越多的作业，这样就会增加系统初始化作业和清理作业的负担。如果数据块太大，数据传输可能需要很长时间才能完成。这也可能导致资源利用不均衡，长时间在一台服务器上运行一个大作业，而其他服务器就会等待而造成处理能力的浪费。不要忘了查看一个任务的作业总数，在必要时调整这个参数。尽量实时监控数据块的传输。

大数据分析结果应该保存成用户期望看到的格式。如果用户要求按照每周的时间序列汇总输出，那么你就要将结果以周为单位进行汇总保存。

第 3 章

◂ 安装与配置大数据软件 ▸

Hadoop 正式诞生于 2006 年 1 月 28 日，是多个开源项目的生态系统，它从根本上改变了企业存储、处理和分析数据的方式。Hadoop 以一种开源的方式创建，开源的强大力量可以创造标准，人人共享，这样才有更多的人参与进来并不断完善。十年前谁也没有料想到 Hadoop 能取得今天这样的成就。Hadoop 之父 Doug Cutting 认为 Hadoop 正处于蓬勃的发展期，而且这样的蓬勃至少还需要几十年。由于 Hadoop 深受客户欢迎，许多公司都推出了各自版本的 Hadoop，也有一些公司则围绕 Hadoop 开发产品。我们首先介绍那些提供 Hadoop 发行版的主流厂商，然后选取其中一个厂商的产品作为示例来安装和配置大数据软件。

3.1 Hadoop 发行版

Hadoop 包含了很多子项目，它们一起构成了 Hadoop 生态圈。在这十年间，新技术（如：Spark）和新版本不断推出，日新月异。这给我们带来 2 个痛点：

- 我们很难及时地跟踪所有这些新技术和新版本；
- 怎么确保这些新旧版本的不同软件组件之间没有冲突。

国外出现了这样的一些公司来解决这些痛点：他们将所有这些版本兼容的技术产品打成一个包，并提供了简单的安装程序和集成管理系统。虽然这些公司采用不同的方式方法，但是都基本解决了上述的痛点。这些公司就是“推出了各自版本的 Hadoop”的公司。

不收费的 Hadoop 版本主要有三个（均是国外厂商），分别是：Apache（最原始的版本，所有发行版均基于这个版本进行改进）、Cloudera 版本（Cloudera’s Distribution Including Apache Hadoop，简称 CDH）、Hortonworks 版本（Hortonworks Data Platform，简称 HDP），下面我们简单介绍后面 2 个版本。

3.1.1 Cloudera

Cloudera 公司于 2008 年在美国硅谷创建，是企业级 Hadoop 技术服务提供商，已经获得了 6.7 亿美元的投资。Cloudera 提供了第一个基于开源 Hadoop 的商业发行版，第一个添加 NoSQL

（HBase）到 Hadoop 平台，第一个在 HDFS 上提供 SQL 查询能力的平台（Impala），第一个将流数据处理能力（Spark）添加到 Hadoop 发行版的厂商。

用户真正在乎基于 Hadoop 的平台和能达到的业务结果，而不是 Hadoop 本身。Hadoop 之初的定位就是一个经济型的深度存储和数据处理平台，我们陆续看到如今大大小小的企业都在用这个平台进行部署，涉及的创新应用也越发广泛。而 Cloudera 提供的 Cloudera Hadoop 发行版（简称 CDH）就是一个稳定的 Hadoop 版本，它简化了 Hadoop 本身的安装和管理，让 Hadoop 使用者省心省力（当然，如你果技术能力强，可以用原生 Hadoop，自己定制，这也会更灵活）。

CDH 的系统架构如图 3-1 所示。截至 2016 年 5 月的最新的版本是 CDH 5.7。它的下载地址为：http://www.cloudera.com/downloads.html。推荐的安装方法是使用 cloudera-manager-installer.bin 安装。我们只要从官网下载 cloudera-manager-installer.bin，然后执行这个 bin 文件，剩下的就是等待下载和安装。

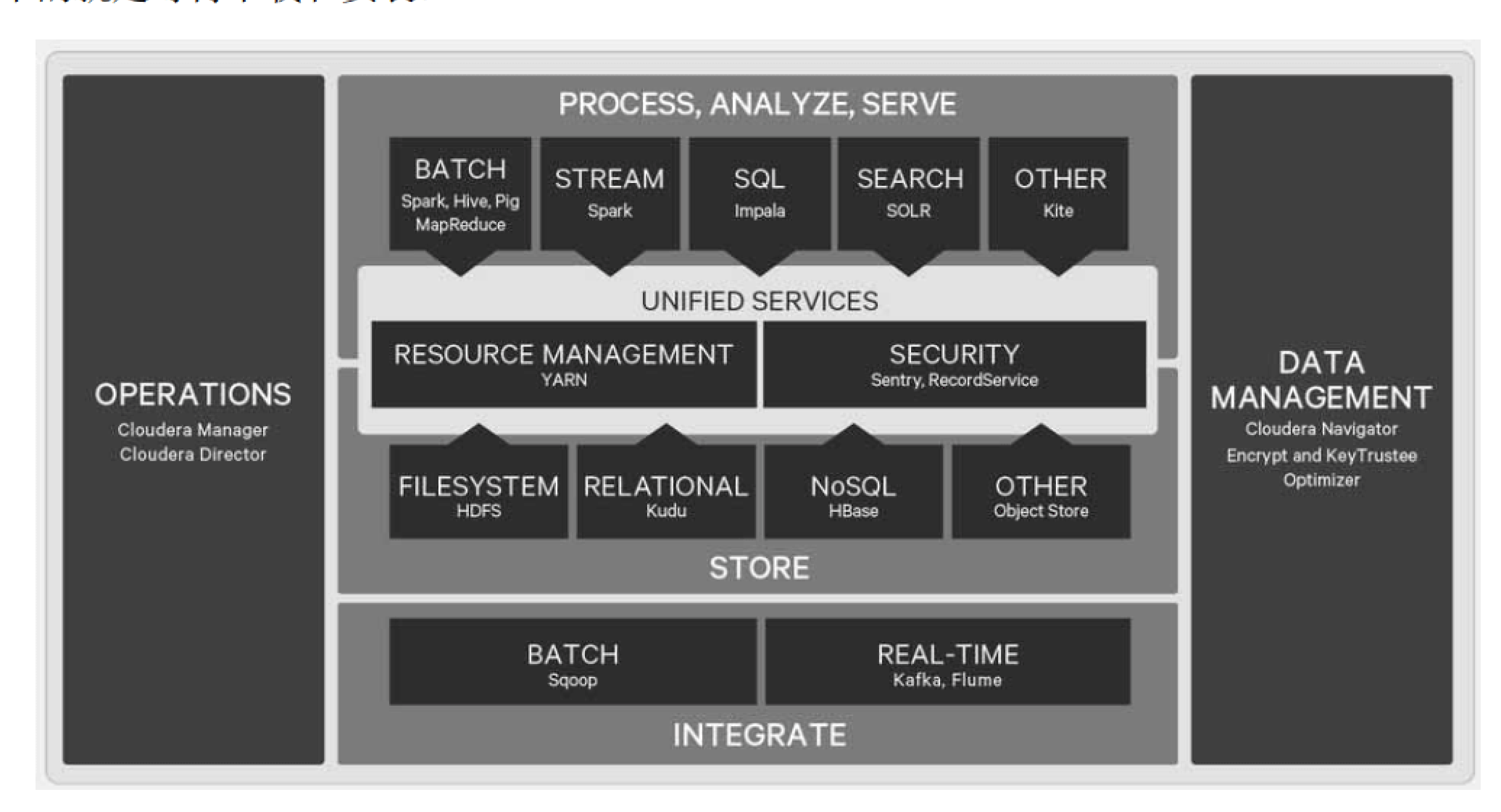


图 3-1　CDH 产品架构　（来自 CDH 官网）

3.1.2　HortonWorks

HortonWorks 公司于 2011 年在美国硅谷创建，已经在 NASDAQ 上市。HortonWorks 提供的 Hadoop 发行版为 Hortonworks Data Platform（HDP）　。HDP 的整个架构如图 3-2 所示。

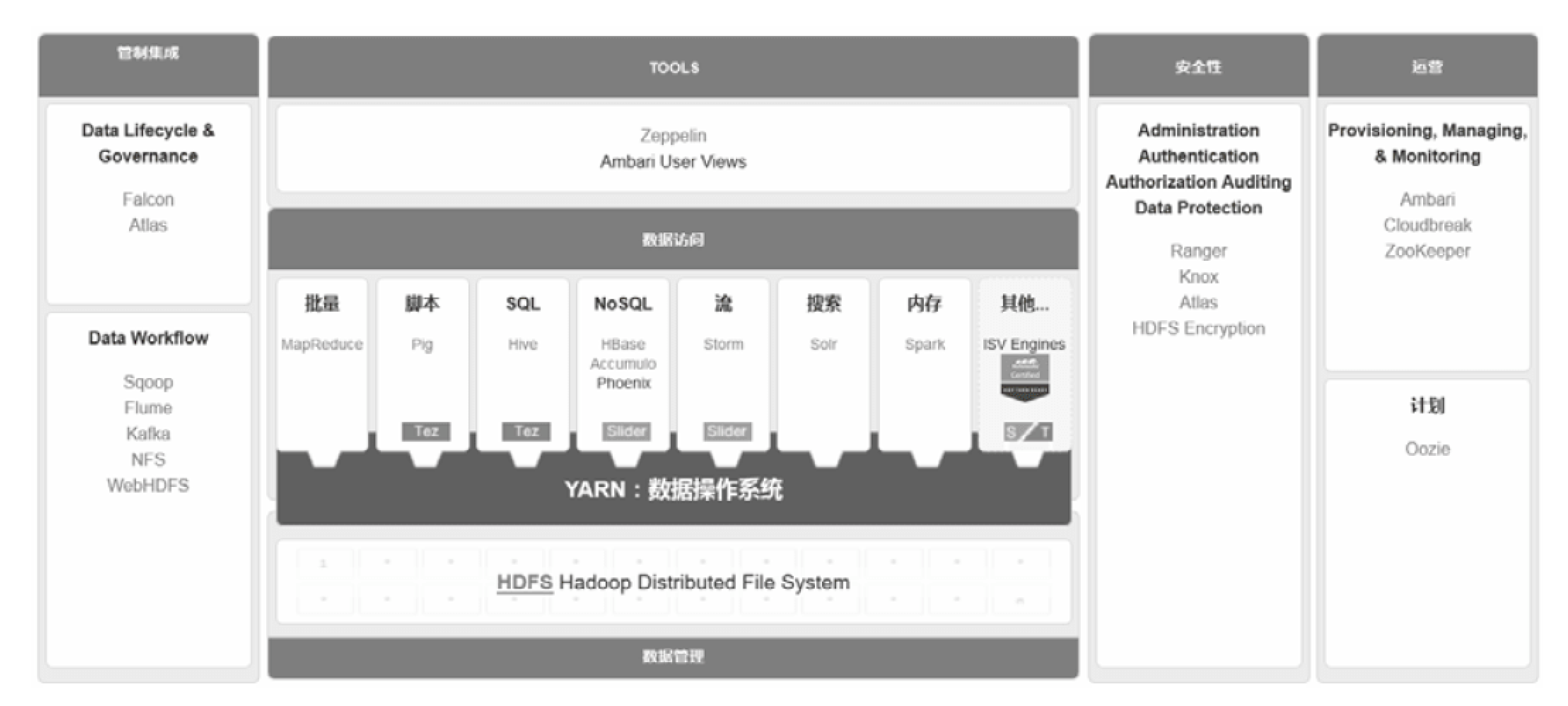


图 3-2　HDP 体系架构

如图 3-2 所示，HDP 包含了 Apache Hadoop 的必要的组件，它包括：YARN、HDFS、Pig、Hive、HBase、Zookeeper 和 Ambari。HDP 还包含了 Apache Spark、Solr 和 Storm 等新兴技术。HDFS 为大数据提供可扩展、容错、具有成本效益的存储。YARN 提供资源管理和可插拔架构，以支持广泛的数据访问方法。YARN 为各种处理引擎提供基础，能够同时以多种方式与相同数据交互（从批量到交互式 SQL 或使用 NoSQL 的低延迟访问）。HDP 能够根据策略加载和管理数据进行身份验证、授权和数据保护。HDP 支持大规模配置、管理、监控和运营 Hadoop 集群。HDP 提供了一整套运营功能，不仅提供集群运行状况的可见性，还提供工具来管理配置。Apache Ambari 提供 API 与现有管理系统集成。HDP 能够与其他的数据分析工具集成。HDP 支持 Windows 系统的安装和配置，并支持以下版本的 Linux：

- RHEL v6.x 和 v5.x
- CentOS v6.x 和 v5.x
- Oracle Linux v6.x 和 v5.x
- SLES v11, SP1 和 SP3
- Ubuntu Precise v12.04

本章后续的安装和配置是以 HDP 为基础进行阐述。

3.1.3　MapR

MapR 也是位于美国硅谷的一个软件公司，专门开发和销售 Apache Hadoop 的衍生软件，它对 Apache Hadoop 主要贡献有：HBase、Pig、Apache Hive 以及 Apache ZooKeeper。MapR 的 Apache Hadoop 发行版提供了完整的数据保护和无单点故障，提高了性能与易用性。MapR 被选择为亚马逊 Elastic Map Reduce（EMR）的升级版本。

MapR 的 MapR Converged Data Platform 如图 3-3 所示。它提供了 2 个版本：免费的社区版（Converged Community Edition）和收费的企业版（Converged Enterprise Edition）。

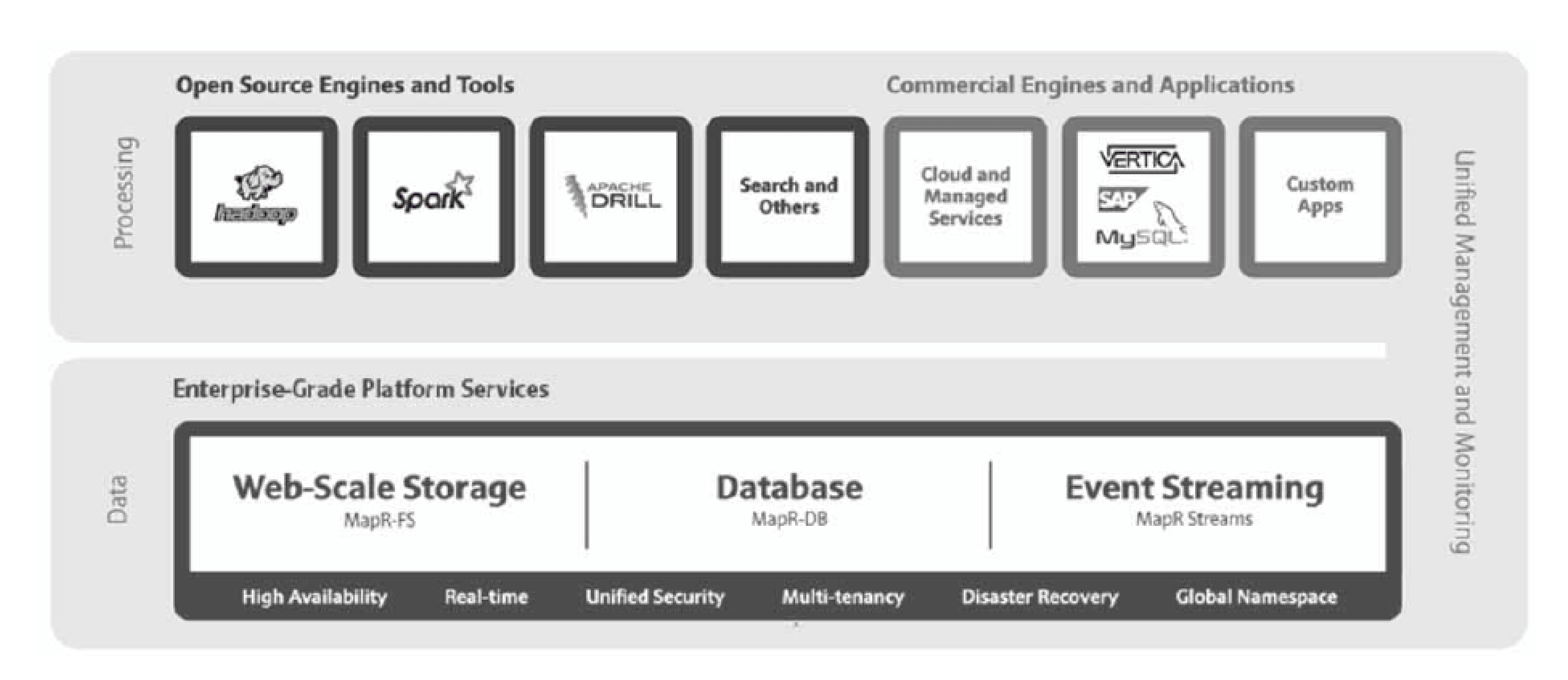


图 3-3　MapR Converged Data Platform

3.2 安装 Hadoop 前的准备工作

在 Hadoop 集群中，大部分的机器都是作为 Datanode 工作的。Datanode 的硬件规格推荐采用表 3-1 的配置：

表 3-1　Datanode 的硬件配置

硬件类型	最低配置
CPU	2 个 4 核 CPU，至少 2-2.5GHz
内存	16-24GB 内存
硬盘	4 个磁盘驱动器（单盘 1-2T）
网络	千兆以太网

Namenode 提供整个 HDFS 文件系统的目录管理、块管理等所有服务，因此需要更多的内存，与集群中的数据块数量相对应，并且需要优化 RAM 的内存通道带宽，采用双通道或三通道以上内存。硬件规格可以采用表 3-2 的配置：

表 3-2　Namenode 的硬件配置

硬件类型	最低配置
CPU	2 个 4 核/8 核 CPU
内存	16-72GB 内存
硬盘	8-12 个磁盘驱动器（单盘 1-2T）
网络	千兆/万兆以太网

Secondary namenode 在小型集群中可以和 Namenode 共用一台机器，较大的集群可以采用与

Namenode 相同的硬件。考虑到关键节点的容错性，建议读者购买加固的服务器来运行 Namenodes 和 Jobtrackers，配有冗余电源和企业级 RAID 磁盘。最好是有一个备用机，当 namenode 或 jobtracker 其中之一突然发生故障时可以替代使用。

Hadoop 集群往往需要运行几十、几百或上千个节点，构建匹配其工作负载的硬件，可以为一个运营团队节省可观的成本，因此，需要精心的策划和慎重的选择。读者需要注意的是，HDFS 目前还不是一个 HA（高可用性）系统，这是因为 NameNode 是 HDFS 集群中的单点失败。如果 NameNode 下线了，那整个 HDFS 文件系统就不可用。虽然我们可以在另一个单独的机器上部署第二个 NameNode，但是这第二个 NameNode 无法做到实时的冗余性，它只是提供了一个有延时的副本。根据我们的实际经验，DataNode 不需要使用 RAID 存储，这是因为文件数据已经在多服务器之间复制了。我们建议 NameNode 所在的机器应该是：

1. 具有很多内存的，性能良好的服务器；内存越多，文件系统越大，块的大小可以越小；
2. 尽量使用 ECC RAM；
3. 不要在 NameNode 所在的机器上安装 DataNode、JobTracker 或 TaskTracker 服务。

上面是整个硬件的一些考虑。 对于软件，特别是操作系统部分，可选的 Linux 很多。表 3-3 是我们在本书中的一个选择，供读者参考。

表 3-3　软件配置

软件类型	推荐配置
OS	64 位 Linux Centos6.5
JDK	jdk-7u45-linux-x64.rpm
openSSL	1.0.1 build 16
python	2.6 或更高
HDP（Hortonworks Data Platform）	2.4
Apache ambari	2.2

HDP2.4 是 HortonWorks 提供的 Hadoop 发行版，提供大数据云存储、大数据处理和分析等服务。Apache Ambari 是对 Hadoop 进行监控、管理和生命周期管理的基于网页的开源项目。Ambari 解决 Hadoop 生态系统部署，这是因为 hadoop 组件间有依赖，包括配置、版本、启动顺序、权限配置等。

3.2.1　Linux 主机配置

为了方便集群中各个主机之间的通信，我们需要设置各主机 IP 地址。我们以两台机器为例来安装和配置 Hadoop，如表 3-4 所示。

表 3-4　主机配置

机器名称	IP 地址	数量
master	192.168.0.110	1
slave01	192.168.0.89	1

修改完成后，输入命令 IP addr 查看 IP 是否修改成功。为了方便集群中各个主机使用机器名称进行通信，我们设置主机名如下：

```
hostname master                  #修改主机名为 master
cat /etc/sysconfig/network          #打开网络配置文件
```

如果输出结果中有“HOSTNAME=master”，则修改成功。按照上面的方法修另外一个 centos 系统的主机名为 slave01。

3.2.2　配置 Java 环境

Hadoop 需要 Java 的支持，下面我们给集群中的各主机配置 Java 环境。

1. 第一步：检查系统是否有已安装好的 jdk。具体操作如下：

```
rpm -qa|grep jdk      #查看已安装的 jdk
```

如果系统已安装 jdk，则需先卸载对应的 jdk，命令如下：

```
rpm -e --nodeps jdk-1.7.0_25-fcs.x86_64   #卸载对应的 jdk
```

具体如下所示：

```
[root@master hadoop]# rpm -qa|grep jdk
jdk-1.7.0_25-fcs.x86_64
[root@master hadoop]# rpm -e --nodeps jdk-1.7.0_25-fcs.x86_64
[root@master hadoop]# java -version
bash: java: command not found
[root@master hadoop]#
```

2. 第二步：下载 JDK

访问 Java 官方网站，如图 3-4 所示，找对图中框线的部分，下载即可。

Java SE Development Kit 7u45

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement　Decline License Agreement

Product / File Description	File Size	Download
Linux ARM v6/v7 Hard Float ABI	67.67 MB	jdk-7u45-linux-arm-vfp-hflt.tar.gz
Linux ARM v6/v7 Soft Float ABI	67.68 MB	jdk-7u45-linux-arm-vfp-sflt.tar.gz
Linux x86	115.62 MB	jdk-7u45-linux-i586.rpm
Linux x86	132.9 MB	jdk-7u45-linux-i586.tar.gz
Linux x64	116.91 MB	jdk-7u45-linux-x64.rpm
Linux x64	131.7 MB	jdk-7u45-linux-x64.tar.gz
Mac OS X x64	183.84 MB	jdk-7u45-macosx-x64.dmg
Solaris x86 (SVR4 package)	139.93 MB	jdk-7u45-solaris-i586.tar.Z
Solaris x86	95.02 MB	jdk-7u45-solaris-i586.tar.gz
Solaris x64 (SVR4 package)	24.6 MB	jdk-7u45-solaris-x64.tar.Z
Solaris x64	16.23 MB	jdk-7u45-solaris-x64.tar.gz
Solaris SPARC (SVR4 package)	139.38 MB	jdk-7u45-solaris-sparc.tar.Z
Solaris SPARC	98.17 MB	jdk-7u45-solaris-sparc.tar.gz
Solaris SPARC 64-bit (SVR4 package)	23.91 MB	jdk-7u45-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	18.26 MB	jdk-7u45-solaris-sparcv9.tar.gz
Windows x86	123.49 MB	jdk-7u45-windows-i586.exe
Windows x64	125.31 MB	jdk-7u45-windows-x64.exe

Back to top

图 3-4　下载 JDK

3. 第三步：安装 JDK

进入 jdk 所在目录，输入以下命令安装 jdk：

```
yum install  jdk-7u45-linux-x64.rpm
```

按照提示，按回车键，即可完成安装。

第四步：配置 Java 环境

```
vim /etc/profile
```

在文件最末尾加上如下信息：

```
export  JAVA_HOME=/usr/java/jdk1.7.0_45
export  PATH=$JAVA_HOME/bin:$PATH
export  CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

第五步：测试 Java 是否安装成功

输入“java –version”，如果看到屏幕上显示如下信息：

```
[root@localhost share]# java -version
java version "1.7.0_45"
Java(TM) SE Runtime Environment (build 1.7.0_45-b18)
Java HotSpot(TM) 64-Bit Server VM (build 24.45-b08, mixed mode)
[root@localhost share]#
```

则证明安装成功。然后按照上面的方法在其他机器上安装 JDK。

3.2.3　安装 NTP 和 python

1. 给集群中各主机安装 NTP

NTP 用来在服务器上保持时间同步。由于集群中的主机时间必须要同步，所以我们必须安

装 NTP 并启动 NTP 服务，输入命令：

```
yum install ntp -y
service ntp start              #开启 ntpd 服务
ntpdate asia.pool.ntp.org
chkconfig ntpd on              #设置 ntpd 服务为默认启动
```

2. 安装 python

由于 ambari 是基于 python 编写的，所以我们必须给集群中各主机安装 python 2.6 或更高的版本。CentOS 6.5 默认已安装 python，我们输入以下命令查看版本是否满足要求（python2.6 or later）：

```
python --version          #查看 python 版本（python2.6 or later）
```

Python 版本信息如图 3-5 所示：

```
[root@centos1 ~]# python --version
Python 2.6.6
[root@centos1 ~]#
```

图 3-5　Python 版本信息

3.2.4　安装和配置 openssl

由于 master 与 slave 之间是通过 SSH 通信的，而 SSH 是依赖于 SSL 的，所以，下面我们给集群中各主机安装或升级 openssl 版本的方法。

1. 第一步：检查 openssl 版本

```
输入： rpm -qa | grep openssl
结果： openssl-1.0.1e-15.el6.x86_64
```

2. 第二步：升级 openssl

上一步中如果输出的是：openssl-1.0.1e-15.x86_64 (1.0.1 build 15) ，则我们需要通过下面的命令行来升级 openssl。

```
yum upgrade openssl
```

3. 第三步：检查 openssl 是否为最新版本（1.0.1 build 16）

```
rpm -qa | grep openssl
```

结果应是：openssl-1.0.1e-16.el6.x86_64。

3.2.5 启动和停止特定服务

1. 第一步：开启 apache

我们还需要开启集群中各主机的 apache http 服务，输入如下命令：

```
service httpd restart
```

2. 第二步：关闭防火墙、SELinux，并统一时区

由于 Hadoop 有一套自身的安全机制，所以我们需要关闭集群中所有服务器的防火墙及 SELinux，并统一时区。

关闭防火墙：

```
chkconfig --level 35 iptables off  #终止防火墙
service iptables stop            #关闭防火墙服务
```

关闭 SELinux：

```
vi /etc/sysconfig/selinux
SELINUX=disabled         #修改后需要重启生效。
setenforce 0         #执行后不需要重启。
```

最后需要统一时间，设置 Linux 系统为同一个时区。

3.2.6 配置 SSH 无密码访问

以下操作只需在 master 上进行。运行本书附录上给出的 auth-ssh.sh 脚本即可：

```
./auth-ssh.sh
```

然后测试是否互通。在 master 上，输入“ssh slave01”。如果出现如下所示的信息，则表示 master 可以免密码登录 slave01 成功：

```
[hadoop@master Desktop]$ ssh slave01
Last login: Tue Feb 18 19:35:56 2014 from master
[hadoop@slave01 ~]$
```

同样，在 slave01 上，输入“ssh master”，如果出现如下所示的信息，表示 slave01 可以免密码登录 master 成功：

```
[hadoop@slave01 Desktop]$ ssh master
Last login: Tue Feb 18 19:35:36 2014 from slave01
[hadoop@master ~]$
```

3.3 安装 Ambari 和 HDP

虽然 Ambari 和 HDP 提供了在线安装，但是由于安装文件很大，所以，我们建议先下载安装文件，然后离线安装。安装过程我们分几个小节说明如下。

3.3.1　配置安装包文件

当安装文件下载后，我们将这些压缩包的文件解压到/var/www/html 中。为了方便管理，我们建议在该目录下创建一个 hdp 子目录，将这些安装包都放在这个目录中。我们使用 tar 命令解压缩：

```
mkdir -p /var/www/html/hdp
tar -xvf ./HDP-2.3.0.0-centos6-rpm.tar.gz -C /var/www/html/hdp/
tar -xvf ./HDP-UTILS-1.1.0.20-centos6.tar.gz -C /var/www/html/hdp/
tar -xvf ./ambari-2.1.0-centos6.tar.gz -C /var/www/html/hdp/
```

之后，在/etc/yum.repos.d 创建三个 repo 文件：



请复制以下 baseurl 的链接地址到浏览器中，看是否能打开。如不能打开，则需要找到对应的文件地址，对 repo 文件的 baseurl 进行修改。

ambari.repo

```
[ambari-2.1.0]
name= ambari-2.1.0
baseurl=http://192.168.0.110/hdp/ambari-2.1.0/centos6/
enabled=1
priority=1
```

hdp.repo

```
[HDP-2.3.0.0]
name=Hortonworks Data Platform Version - HDP-2.3.0.0
baseurl= http://192.168.0.110/hdp/hdp/centos6/2.x/GA/2.3.0.0
enabled=1
priority=1
```

hdp-util.repo

```
[HDP-UTILS-1.1.0.20]
name=Hortonworks Data Platform Version - HDP-UTILS-1.1.0.20
baseurl= http://192.168.0.110/hdp/hdp-util/repos/centos6
enabled=1
priority=1
```

之后，将写好的文件，发送至其他节点上：

```
scp ambari.repo slave01:/etc/yum.repo.d/
scp hdp.repo slave01:/etc/yum.repo.d/
scp hdp-util.repo slave01:/etc/yum.repo.d/
```

3.3.2 安装 Ambari

将 repo 文件发送至各节点后，在各节点需要运行 yum clean all 以清空缓存文件，为了检验文件是否配置正确，可以使用 yum search ambari-agent、yum search Oozie、yum search gangli 命令检查。如果配置有问题，就会出现找不到文件包的问题。

在主节点运行：

```
yum install ambari-server
```

在所有节点上运行：

```
yum install ambari-agent
```

yum 是一个在 Shell 上使用的软件包管理器。

基于我们的经验，在安装 Ambari 时，有时出现下面错误：

```
rpmts_HdrFromFdno: Header V4 RSA/SHA1 Signature, key ID 07513cad:
NOKEY
Public key for ambari-server-2.1.0-1470.x86_64.rpm is not installed
```

解决办法如下：

（1）将 RPM-GPG-KEY-Jenkins 放入/etc/pki/rpm-gpg

（2）运行 rpm --import /etc/pki/rpm-gpg/RPM*

在主节点上，运行以下命令启动 Ambari 服务器：

```
ambari-server start
```

在所有节点上，运行命令启动 ambari agent：

```
ambari-agent start
```

在所有节点上，修改 /etc/ambari-agent/conf/ambari-agent.ini 文件：

```
vi  /etc/ambari-agent/conf/ambari-agent.ini

[server]
hostname=master   #注意：hostname 为主节点的主机名
```

打开浏览器，输入地址：

```
http://master:8080
```

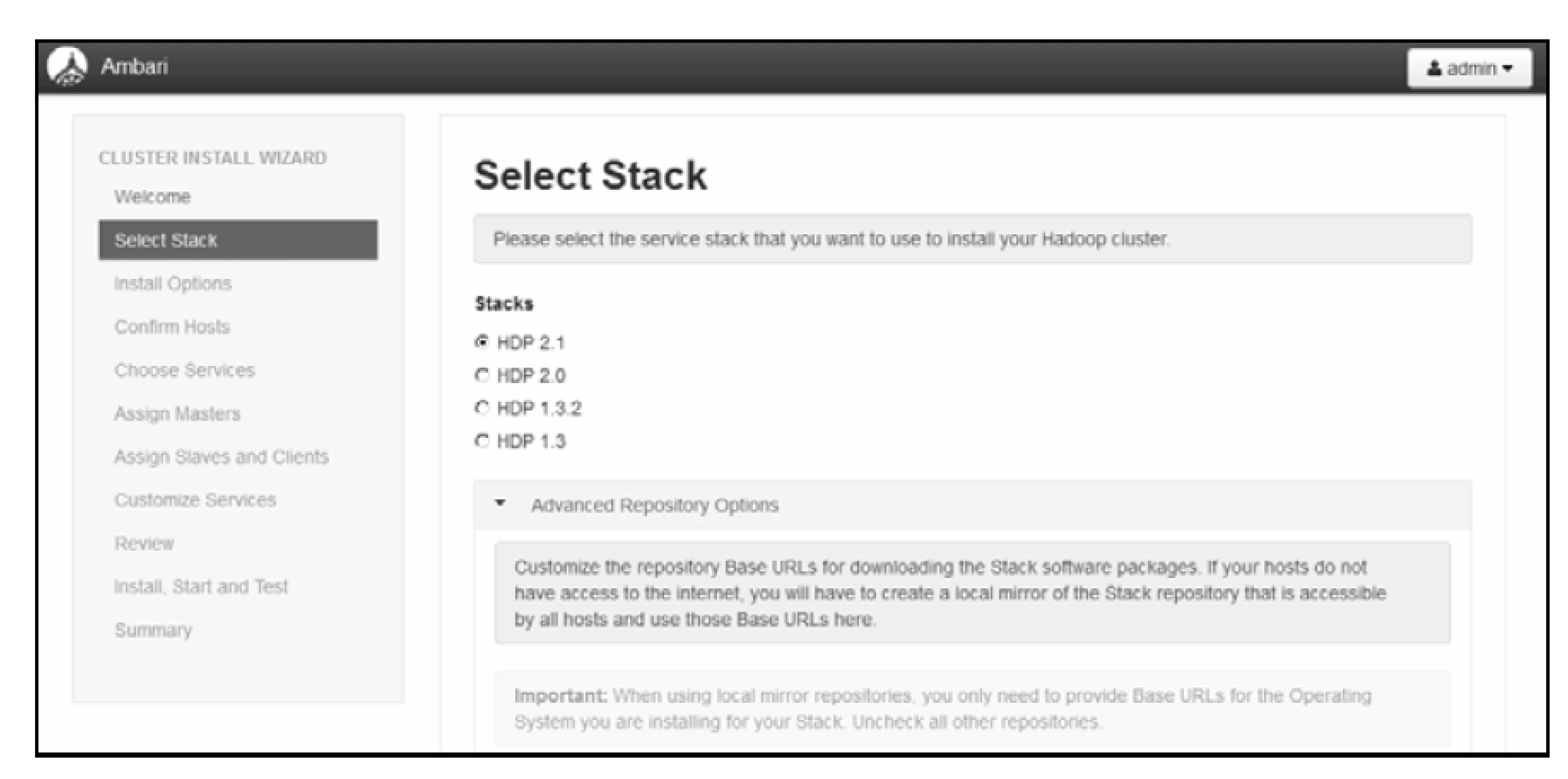
出现 Ambari 的登录界面，登录的用户名和密码为：

```
用户名：admin
密码：admin
```

3.3.3　安装和配置 HDP

登录 Ambari 后，进行配置向导，这时就可以按照自己的需要进行 HDP 的安装和配置了。具体步骤如下：

步骤 01　选择版本，如图 3-6 所示：



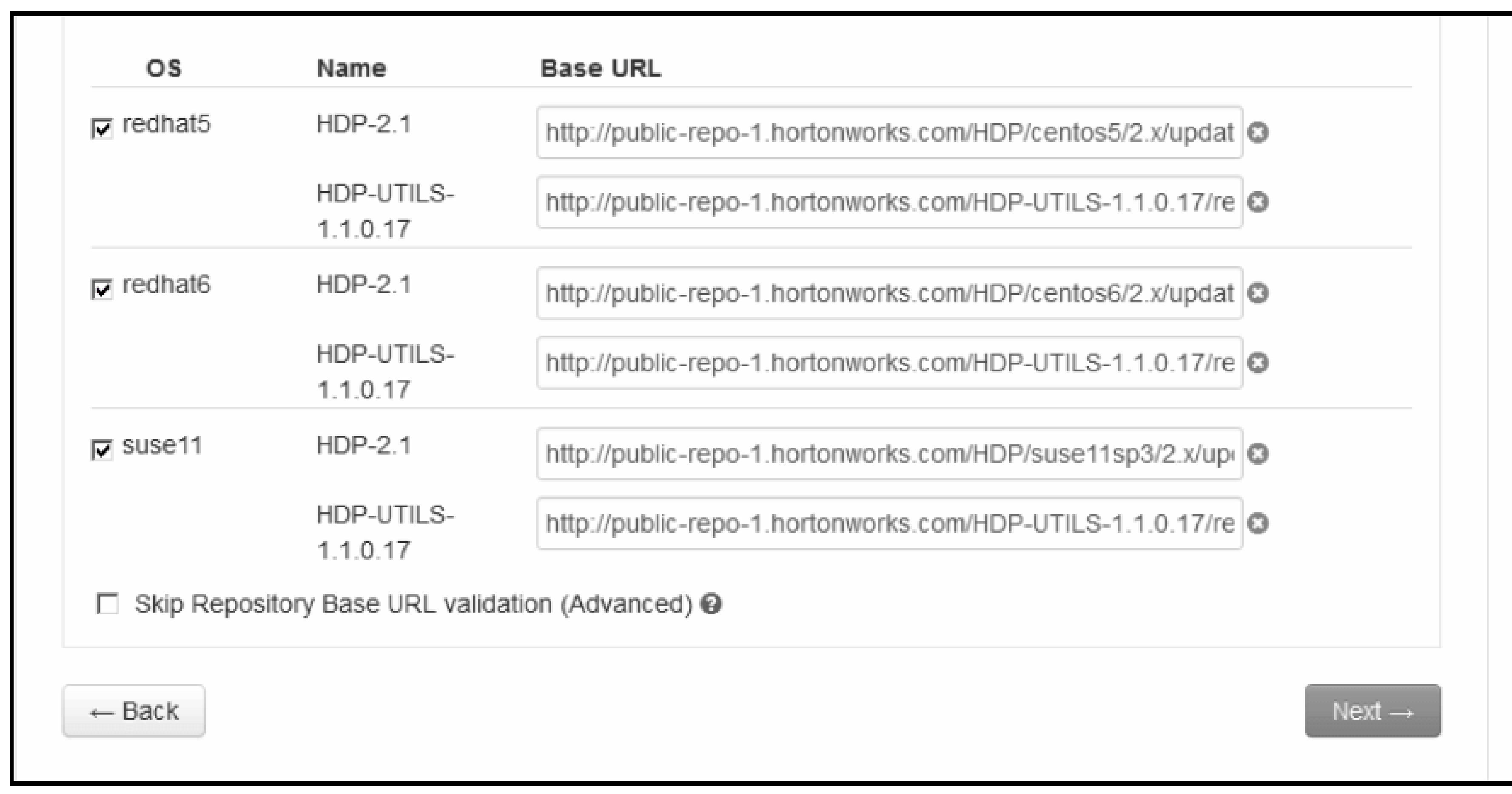


图 3-6　选择安装版本

在 OS 上，只选择 redhat6 一栏。我们推荐使用本地安装，因此在这里我们需要修改对应的 yum 源地址。我们将后面的 Base URL 改为如下地址：

```
Hdp-2.3.0 http://192.168.0.110/hdp/hdp/centos6/2.x/GA/2.3.0.0
Hdp-util http://192.168.0.110/hdp/hdp-util/repos/centos6
```

注意：请直接把地址复制到浏览器中，看是否可以访问。

步骤 02　单击 Next 按钮，在 Install Options 上配置 SSH 键，如图 3-7 所示。

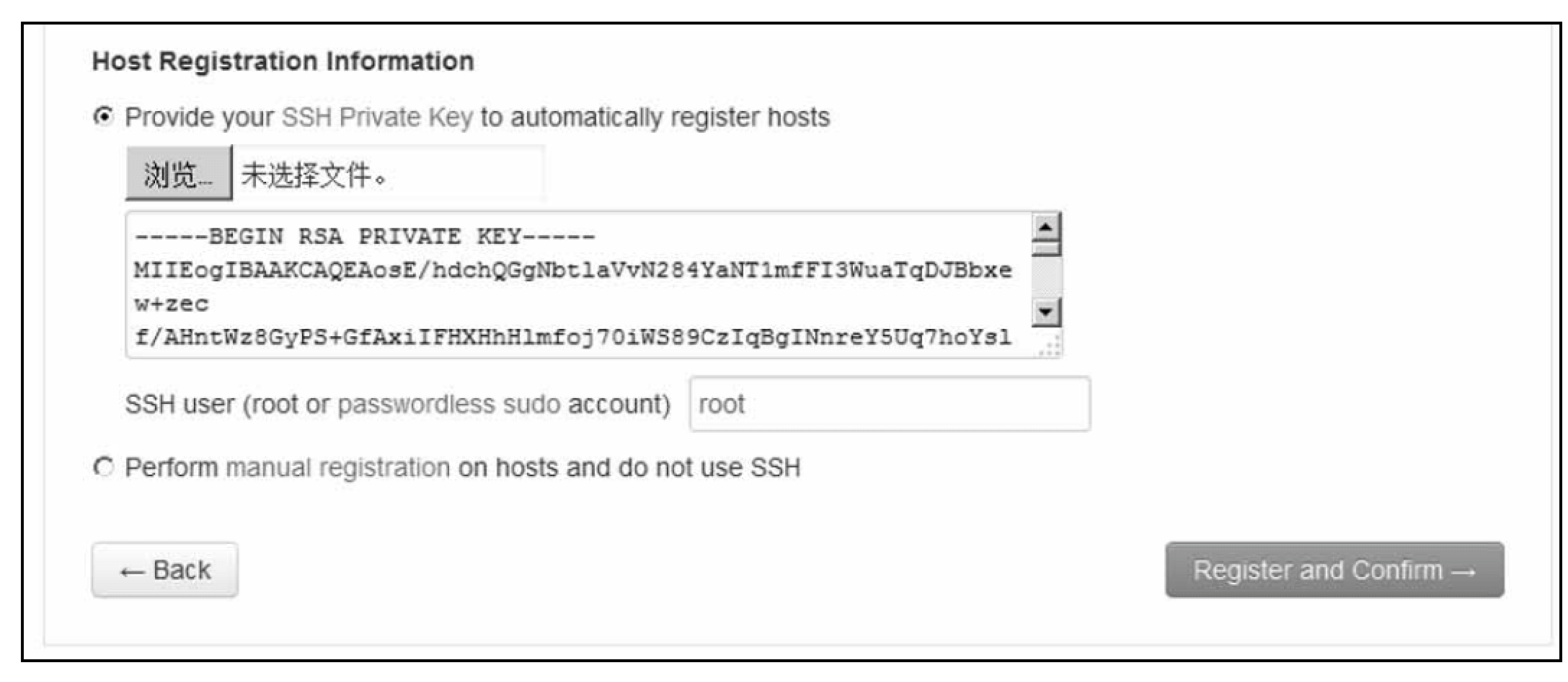


图 3-7　设置 SSH 键

我们可以通过下面的方法获得 SSH private key：

```
cd ~/.ssh              #进入 ssh 目录
cat id_rsa             #获取 SSH private key 内容
```

然后拷贝上述命令的输出结果到 Install Options 窗口中。

步骤 03　在“Confirm Hosts”中确认节点，然后点击 Next 按钮。

步骤 04　在“Choose Services”窗口确认安装的服务，选择默认值即可，如图 3-8 所示。

步骤 05　在“Assign Masters”窗口确认安装的 Master 的服务，选择默认值即可。

步骤 06　在“Assign Slaves and Clients”窗口确认安装的 Slave 的服务，选择默认值即可。

步骤 07　最后确认安装的服务版本，就开始安装。

步骤 08　安装结束后，安装程序会给出总结信息。

步骤 09　安装成功后的界面如图 3-9 所示。

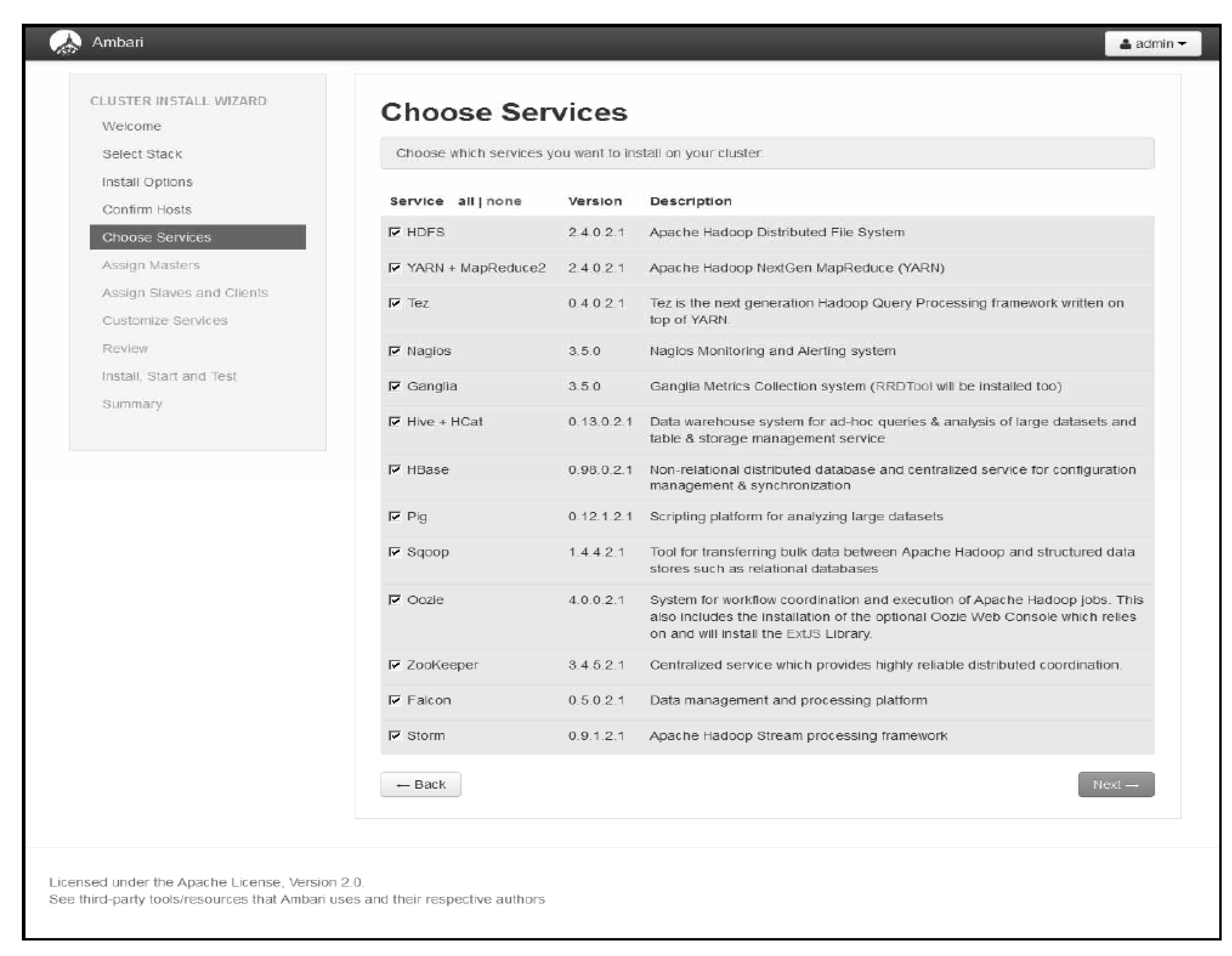


图 3-8　选择安装组件

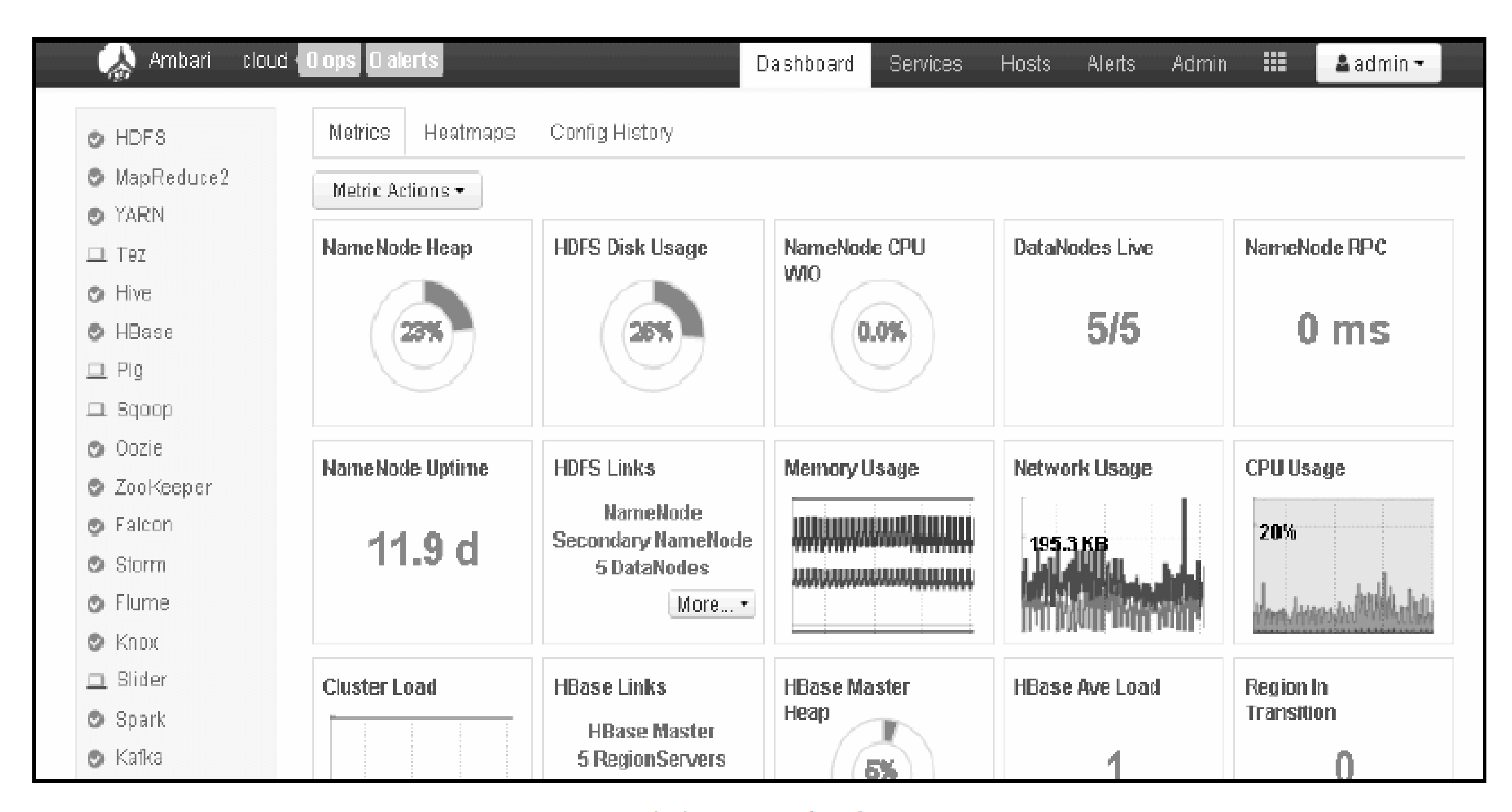


图 3-9　启动 HDP

3.4 初识 Hadoop

在 Hadoop 安装后，我们启动 Hadoop 相关服务，然后尝试使用这些服务。

3.4.1 启动和停止服务

进入 ambari 后，可以看到如图 3-10 所示的界面。左侧是 HDP 包含的所有组件，如果组件左侧显示为绿色对号，表示成功启动。

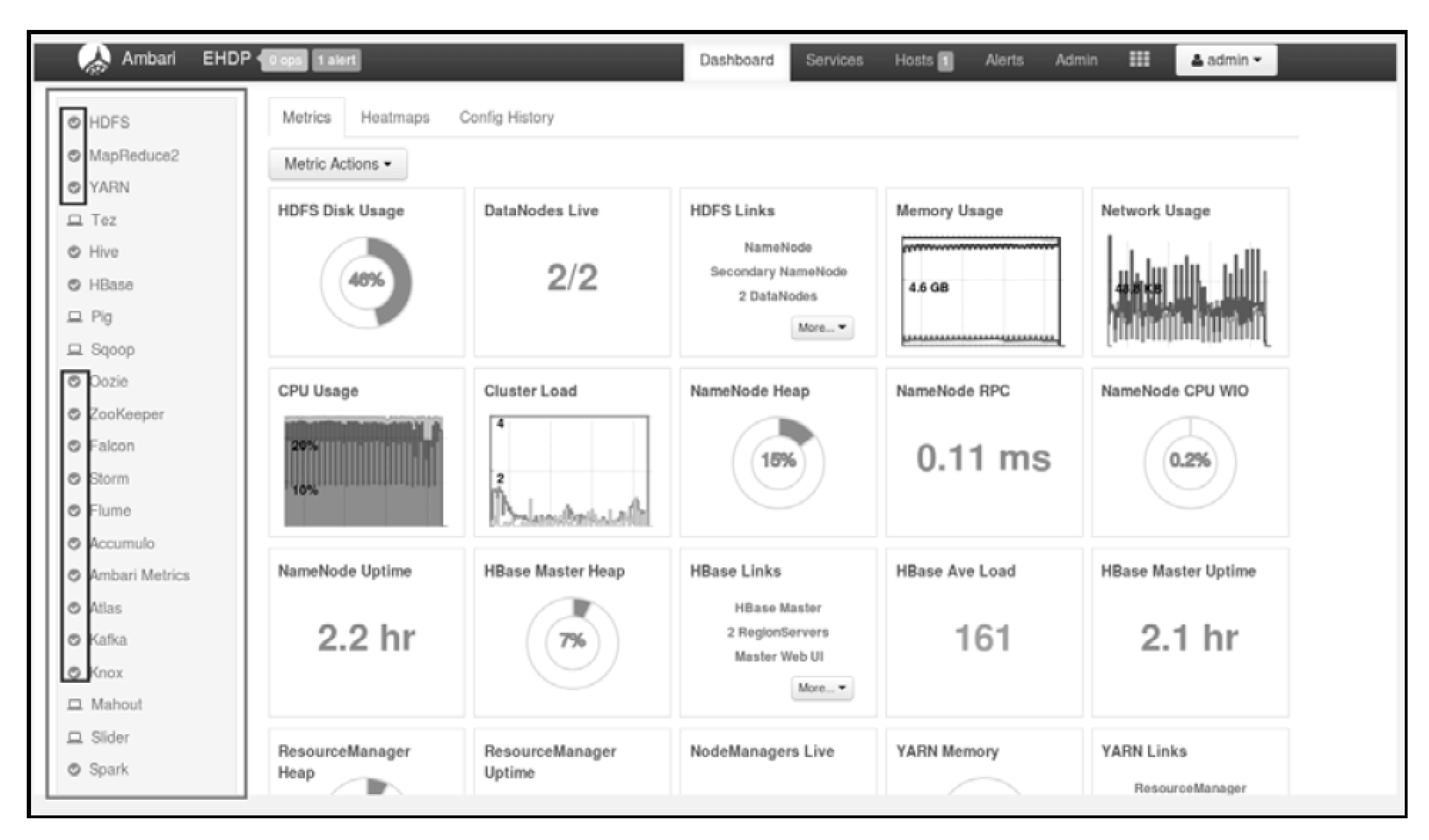


图 3-10　ambari 界面

以 HDFS 服务为例，我们点击右上角，选择对应的选项，启动或停止服务即可，如图 3-11 所示。

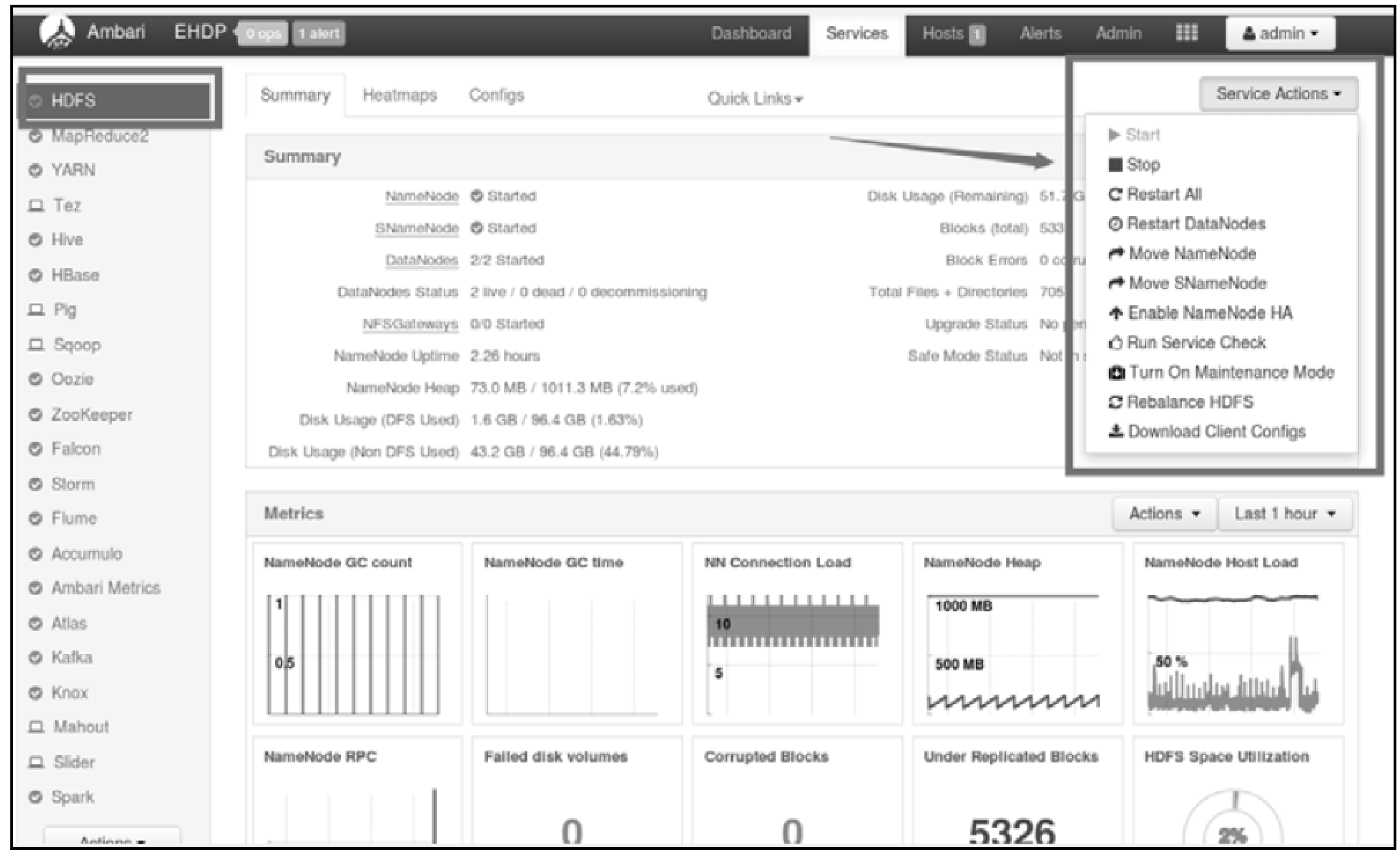


图 3-11　HDFS 管理

3.4.2　使用 HDFS

我们对 HDFS 做一些简单的测试操作：首先查看 HDFS 状态，查看有哪些 datanode，以及各个 datanode 的情况。我们输入以下命令：

```
sudo -u hdfs hdfs dfsadmin -report
```

上面的“sudo -u hdfs”是切换到 hdfs 用户，而 dfsadmin 是运行一个 HDFS 的 dfsadmin 客户端。它的参数–report 用来报告文件系统的基本信息和统计信息，如图 3-12 所示。

```
[root@master Desktop]# sudo -u hdfs hdfs dfsadmin -report
Configured Capacity: 103541891072 (96.43 GB)
Present Capacity: 57167847382 (53.24 GB)
DFS Remaining: 55480068418 (51.67 GB)
DFS Used: 1687778964 (1.57 GB)
DFS Used%: 2.95%
Under replicated blocks: 5416
Blocks with corrupt replicas: 0
Missing blocks: 0
Missing blocks (with replication factor 1): 0

-------------------------------------------------
Live datanodes (2):

Name: 192.168.0.89:50010 (slave01)
Hostname: slave01
Decommission Status : Normal
Configured Capacity: 51770945536 (48.22 GB)
DFS Used: 843891530 (804.80 MB)
Non DFS Used: 14474557461 (13.48 GB)
DFS Remaining: 36452496545 (33.95 GB)
DFS Used%: 1.63%
DFS Remaining%: 70.41%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
Xceivers: 12
Last contact: Mon Aug 24 19:08:22 CST 2015


Name: 192.168.0.110:50010 (master)
Hostname: master
Decommission Status : Normal
Configured Capacity: 51770945536 (48.22 GB)
DFS Used: 843887434 (804.79 MB)
Non DFS Used: 31899486229 (29.71 GB)
DFS Remaining: 19027571873 (17.72 GB)
DFS Used%: 1.63%
DFS Remaining%: 36.75%
Configured Cache Capacity: 0 (0 B)
Cache Used: 0 (0 B)
Cache Remaining: 0 (0 B)
Cache Used%: 100.00%
Cache Remaining%: 0.00%
```

图 3-12　显示 HDFS 状态

下面我们在 HDFS 上创建一个文件夹，输入命令：

```
hadoop fs -mkdir /tmp/input
```

并将本地文件 input1.txt 传到 hdfs 的/tmp/input 目录下：

```
hadoop fs -put '/root/Desktop/input1.txt' /tmp/input
```

然后查看 HDFS 上的文件，验证各节点的 input1.txt 是否上传成功：

```
hadoop fs -ls /tmp/input
```

以上操作命令执行结果如图 3-13 所示。

```
[root@master Desktop]# hadoop fs -mkdir /tmp/input
mkdir: `/tmp/input': File exists
[root@master Desktop]# hadoop fs -put '/root/Desktop/input1.txt' /tmp/input
[root@master Desktop]# hadoop fs -ls /tmp/input
Found 1 items
-rw-r--r--   3 root hdfs          0 2015-08-24 19:18 /tmp/input/input1.txt
[root@master Desktop]# ssh slave01
Last login: Mon Aug 24 17:39:15 2015 from master
[root@slave01 ~]# hadoop fs -ls /tmp/input
Found 1 items
-rw-r--r--   3 root hdfs          0 2015-08-24 19:18 /tmp/input/input1.txt
[root@slave01 ~]# █
```

图 3-13　文件操作

读者可以在 HDFS 上执行以下更多的命令,对文件进行操作：

```
hadoop fs -get  input1.txt /tmp/input/input1.txt  #把 HDFS 文件拉到本地
hadoop fs -cat /tmp/input/input1.txt  #查看 HDFS 上的文件
```

3.5 Hadoop 的特性

Hadoop 是一个能够让用户轻松架构和使用的分布式计算平台。用户可以轻松地在 Hadoop 上开发和运行处理海量数据的应用程序。它主要有以下几个优点：

- 高扩展性：Hadoop 可以扩展至数千个节点，对数据持续增长，数据量特别巨大的需求很合适。
- 高效：Hadoop 能够在节点之间动态地移动数据，并保证各个节点的动态平衡，因此处理速度非常快。
- 高容错性：Hadoop 能够自动保存数据的多个副本，并且能够自动将失败的任务重新分配。
- 低成本：Hadoop 是开源项目，不仅从软件上节约成本，而且 hadoop 对硬件上的要求也不高，因此也从硬件上节约了一大笔成本。

第 4 章

大数据存储：文件系统

一旦数据进入大数据系统，清洗并转化为所需格式时，都需要将数据存储到一个合适的持久层中。一个最常见的大数据存储的地方就是文件系统。HDFS 是 Hadoop 分布式文件系统的简称，是被设计成适合运行在通用硬件上的分布式文件系统。它和现有的分布式文件系统有很多共同点：

（1）它对存储空间进行统一管理。在用户创建新文件时为其分配空闲空间，在用户删除或修改某个文件时，回收和调整存储空间；

（2）它实现了按名存取和透明存取。所谓透明存取是指不必了解文件存放的物理结构和查找方法等与存取介质有关的部分；

（3）提供文件和文件夹创建、更新和删除的功能。

HDFS 是支撑大文件的系统，典型的 HDFS 文件大小是 GB 到 TB 的级别。所以，HDFS 采用了流式数据访问技术。大数据分析经常读取一个海量数据集的大部分数据甚至全部数据，因此读取整个数据集的时间延迟比读取第一条记录的时间延迟更重要。而流式读取最小化了硬盘的寻址开销，只需要寻址一次，然后就一直读（与流数据访问对应的是随机数据访问，它要求定位、查询或修改数据的延迟较小，比较适合于创建数据后再多次读写的情况，传统关系型数据库很符合这一点）。

正如其他的文件系统，我们可以通过多种方式访问和管理 HDFS 上的文件和目录：HDFS 为开发人员提供 Java API；可以在一个 HTTP 浏览器中浏览 HDFS 中的文件；可以使用 Hadoop shell 命令来访问文件系统。

4.1 HDFS shell 命令

HDFS 提供了众多的 shell 命令来访问和管理 HDFS 上的文件。Hadoop 自带的 shell 脚本为 hadoop，对于 HDFS 的 shell 命令，就是使用“Hadoop fs -命令”的格式来执行，如图 4-1 所示。

```
[root@master01 ~]# hadoop fs
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] [-l] <localsrc> ... <dst>]
        [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] [-h] [-v] [-t [<storage type>]] <path> ...]
        [-cp [-f] [-p | -p[topax]] <src> ... <dst>]
```

图 4-1 Hadoop fs -命令

下面是获取 cp shell 命令的帮助信息，如图 4-2 所示。

```
[root@master01 ~]# hadoop fs -help cp
-cp [-f] [-p | -p[topax]] <src> ... <dst> :
  Copy files that match the file pattern <src> to a destination.  When copying
  multiple files, the destination must be a directory. Passing -p preserves status
  [topax] (timestamps, ownership, permission, ACLs, XAttr). If -p is specified
  with no <arg>, then preserves timestamps, ownership, permission. If -pa is
  specified, then preserves permission also because ACL is a super-set of
  permission. Passing -f overwrites the destination if it already exists. raw
  namespace extended attributes are preserved if (1) they are supported (HDFS
  only) and, (2) all of the source and target pathnames are in the /.reserved/raw
  hierarchy. raw namespace xattr preservation is determined solely by the presence
  (or absence) of the /.reserved/raw prefix and not by the -p option.
[root@master01 ~]#
```

图 4-2 Hadoop fs -help cp 命令

下面我们在 HDFS 上创建一个新目录 /yunsheng，如图 4-3 所示。

```
[root@master01 ~]# su hdfs
[hdfs@master01 root]$ ls
ls: cannot open directory .: Permission denied
[hdfs@master01 root]$ pwd
/root
[hdfs@master01 root]$ hadoop fs -mkdir /yunsheng
[hdfs@master01 root]$
```

图 4-3 创建一个新目录命令

然后查看一下这个目录，如图 4-4 所示。

```
[hdfs@master01 root]$ hadoop fs -ls /
Found 13 items
drwxrwxrwx   - yarn   hadoop          0 2015-10-10 09:18 /app-logs
drwxr-xr-x   - hdfs   hdfs            0 2015-09-11 15:59 /apps
drwxr-xr-x   - hdfs   hdfs            0 2015-09-18 11:50 /data
drwxrwxrwx   - hdfs   hdfs            0 2015-10-01 00:17 /flume2
drwxrwxrwx   - hdfs   hdfs            0 2015-09-25 14:33 /flume3
drwxrwxrwx   - hdfs   hdfs            0 2015-10-30 09:50 /flumem10
drwxr-xr-x   - hdfs   hdfs            0 2015-09-08 14:19 /hdp
drwxr-xr-x   - mapred hdfs            0 2015-09-08 14:19 /mapred
drwxrwxrwx   - mapred hadoop          0 2015-09-08 14:19 /mr-history
drwxrwxrwx   - hdfs   hdfs            0 2015-11-04 15:13 /product
drwxrwxrwx   - hdfs   hdfs            0 2015-09-08 14:23 /tmp
drwxrwxrwx   - hdfs   hdfs            0 2015-09-22 15:47 /user
drwxr-xr-x   - hdfs   hdfs            0 2015-11-05 17:50 /yunsheng
[hdfs@master01 root]$
```

图 4-4 查看目录命令

使用 copyFromLocal 命令拷贝一个本地文件到 HDFS 目录下（也可以用 put 命令），并列出 HDFS 目录下的内容，如图 4-5 所示。

```
[hdfs@master01 ~]$ ls
1.txt  6.txt  7.txt  8.txt  9.txt  product11-2
[hdfs@master01 ~]$ hadoop fs -copyFromLocal 9.txt /yunsheng
[hdfs@master01 ~]$ hadllp fs -ls /yunsheng
bash: hadllp: command not found
[hdfs@master01 ~]$ hadoop fs -ls /yunsheng
Found 1 items
-rw-r--r--   3 hdfs hdfs     440055 2015-11-05 18:08 /yunsheng/9.txt
[hdfs@master01 ~]$
```

图 4-5　copyFromLocal 命令

使用 copyToLocal 命令拷贝 HDFS 文件到本地文件夹，也可以使用 get 命令，如图 4-6 所示。

```
[hdfs@master01 ~]$ hadoop fs -copyToLocal /yunsheng/9.txt /home/hdfs/fromHDFS.txt
15/11/05 18:28:39 WARN hdfs.DFSClient: DFSInputStream has been closed already
[hdfs@master01 ~]$ ls -l /home/hdfs/
total 54808
-rw-r--r-- 1 root root    1457012 Sep 28 19:34 1.txt
-rw-r--r-- 1 root root    2772294 Oct 10 10:45 6.txt
-rw-r--r-- 1 root root    5406510 Oct 10 10:45 7.txt
-rw-r--r-- 1 root root    9433368 Oct 10 10:45 8.txt
-rw-r--r-- 1 root root     440055 Oct 10 10:45 9.txt
-rw-r--r-- 1 hdfs hadoop   440055 Nov  5 18:28 fromHDFS.txt
-rw-r--r-- 1 root root   36162400 Nov  2 18:14 product11-2
[hdfs@master01 ~]$
```

图 4-6　copyToLocal 命令

查看 HDFS 文件内容。如图 4-7 所示。

```
PSG1208-K1;A1;901002226;8CAB8E17FBB2;1Hop8;2015-10-09;
PSG1208-K1;A1;901002226;8CAB8E17FBB3;1Hop8;2015-10-09;
PSG1208-K1;A1;901002226;8CAB8E17FBB4;1Hop8;2015-10-09;
PSG1208-K1;A1;901002226;8CAB8E17FBB5;1Hop8;2015-10-09;
PSG1208-K1;A1;901002226;8CAB8E17FBB6;1Hop8;2015-10-09;
PSG1208-K1;A1;901002226;8CAB8E17FBB7;1Hop8;2015-10-09;
[hdfs@master01 ~]$ hadoop fs -cat /yunsheng/9.txt
```

图 4-7　查看 HDFS 文件内容命令

要注意的是，HDFS 中文件可能是几个 TB 的。当拷贝 HDFS 文件到本地文件系统时，要保证本地文件系统有足够的可用空间和较好的网络速度。如果要在两个集群之间进行数据复制，则可以使用 distcp 命令。

4.2 HDFS 配置文件

HDFS 是存储和管理大数据文件的文件系统，它将一个大文件切割成一个一个的数据块，再将这些数据块分发到集群上。我们可以在 Hadoop 安装目录的 conf 文件夹下找到 hdfs-size.xml，如图 4-8 所示。它是 HDFS 配置文件。修改其中的 dfs.block.size 属性值就改变了 HDFS 的块大

小，默认值为 64MB（这么大的数据块可以在硬盘上连续进行存储，这样就保证了以最少的磁盘寻址次数来进行写入和读取，从而最大化提高读写性能）。新的块大小不会对 HDFS 上已有的文件产生影响，新的大小只会影响新传文件的块大小。

```
[root@master01 conf]# pwd
/usr/hdp/2.3.0.0-2557/hadoop/conf
[root@master01 conf]# more hdfs-site.xml
<!--Fri Nov  6 11:10:42 2015-->
    <configuration>

    <property>
      <name>dfs.block.access.token.enable</name>
      <value>true</value>
    </property>

    <property>
      <name>dfs.blockreport.initialDelay</name>
      <value>120</value>
    </property>

    <property>
      <name>dfs.blocksize</name>
      <value>134217728</value>
    </property>
```

图 4-8　hdfs-size.xml

HDFS 通过将数据块复制多份来实现容错性。复制因子是用 HDFS 配置文件的 dfs.replication 属性来设置，如图 4-9 所示。

```
    <property>
      <name>dfs.replication</name>
      <value>3</value>
    </property>

    <property>
      <name>dfs.replication.max</name>
      <value>50</value>
    </property>
```

图 4-9　复制因子

如上图所示，复制因子默认为 3（1 个原始数据块，2 个副本数据块）。直接修改上述属性值将会改变所有上传到 HDFS 文件的默认复制份数。我们也可以使用 shell 命令来设置复制因子。通过 shell 命令可改变一个文件或某一个文件夹下所有文件的复制因子。比如如图 4-10 所示的例子。

```
[hdfs@master01 ~]$ hadoop fs -setrep -w 3 -R /yunsheng
setrep: `-R': No such file or directory
Replication 3 set: /yunsheng/9.txt
Waiting for /yunsheng/9.txt ... done
[hdfs@master01 ~]$
```

图 4-10　改变复制因子

4.3 HDFS API 编程

HDFS 提供了 FileSystem API 在 HDFS 上创建和写入一个文件，或者从 HDFS 文件上读取文件内容。前面章节中介绍的 HDFS shell 命令都是构建在 HDFS FileSystem API 之上的。表 4-1 给出常用的 HDFS 相关类。

表 4-1　常用的 HDFS 相关类

Hadoop 类	功能
org.apache.hadoop.fs.FileSystem	一个通用文件系统的抽象基类，可以被分布式文件系统继承。所有使用 Hadoop 文件系统的代码都要使用到这个类
org.apache.hadoop.fs.FileStatus	客户端可见的文件状态信息
org.apache.hadoop.fs.FSDataInputStream	文件输入流，用于读取 Hadoop 文件
org.apache.hadoop.fs.FSDataOutputStream	文件输出流，用于写 Hadoop 文件
org.apache.hadoop.fs.permission.FsPermission	文件或者目录的权限
org.apache.hadoop.conf.Configuration	访问配置项。所有的配置项的值，如果没有专门配置，以 core-default.xml 为准；否则以 core-site.xml 中的配置为准

4.3.1　读取 HDFS 文件内容

读取 HDFS 文件内容的代码可以分为四个大步骤：获取文件系统对象、通过文件系统打开文件、将文件内容输出，以及关闭对象实例。

1. 获取文件系统对象

HDFS 本身就是一个文件系统，所以要从 HDFS 上读取文件，必须先得到一个 org.apache.hadoop.fs.FileSystem 对象。FileSystem 是一个抽象类，大多数文件系统访问和操作都可以通过这个类的对象来完成。我们通过 FileSystem.get()来得到一个 HDFS 文件系统对象之后，就可以对 HDFS 进行相关操作。

在获取文件系统之前需要读取配置文件，然后才能获取文件系统。读取配置文件是通过 Configuration 类完成的。这个类有三个构造器，无参数的构造器表示直接加载默认资源，也可以指定一个 boolean 参数来关闭加载默认值，或直接使用另外一个 Configuration 对象来初始化。open()方法中指定所要读取的文件路径信息（URI 格式），以“hdfs://”开头。例子如下：

```
package youngPackage.hdfs;
import java.io.BufferedInputStream;
```

```
import java.io.ByteArrayInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.URI;
import java.util.Arrays;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.BlockLocation;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;
import org.apache.hadoop.util.Progressable;

public class HDFSUtil {
public static void downloadFromHdfs(String location, String hdfsPath)
{
    FileSystem fs = null;
    FSDataInputStream fsin = null;
    OutputStream out = null;
    try {
        Configuration conf = new Configuration();//读取配置文件
        fs = FileSystem.get(URI.create(hdfsPath), conf); //获取文件系统
```

2. 通过文件系统打开文件

打开文件其实就是创建一个文件输入流，之后就可以使用输入流对象读取文件中的内容了。

```
fsin = fs.open(new Path(hdfsPath));//打开文件
```

如果我们跟踪 open 方法的代码，我们会发现，这个方法会调用一个名叫 openInfo()方法，openInfo()方法是一个线程安全的方法，作用是从 namenode 获取已打开的文件信息。有兴趣的读者可以读一下如下的 openInfo()源代码。

```
/**
```

```
     * Grab the open-file info from namenode
     */
    synchronized void openInfo() throws IOException, UnresolvedLinkException {
        lastBlockBeingWrittenLength = fetchLocatedBlocksAndGetLastBlockLength();
        int retriesForLastBlockLength = dfsClient.getConf().retryTimesForGetLas
tBlockLength;
        while (retriesForLastBlockLength > 0) {
            // Getting last block length as -1 is a special case. When cluster
             // restarts, DNs may not report immediately. At this time partial b
lock
             // locations will not be available with NN for getting the length.
Lets
            // retry for 3 times to get the length.
            if (lastBlockBeingWrittenLength == -1) {
               DFSClient.LOG.warn("Last block locations not available. "
                  + "Datanodes might not have reported blocks completely."
                  + " Will retry for " + retriesForLastBlockLength + " times");
               waitFor(dfsClient.getConf().retryIntervalForGetLastBlockLength);
               lastBlockBeingWrittenLength = fetchLocatedBlocksAndGetLastBlockLe
ngth();
            } else {
               break;
            }
            retriesForLastBlockLength--;
        }
        if (retriesForLastBlockLength == 0) {
          throw new IOException("Could not obtain the last block locations.");
        }
    }
```

上面的方法调用 fetchLocatedBlocksAndGetLastBlockLength()方法获取块的位置信息。

3. 将文件内容输出

因为之前已经获得了一个 FSDataInputStream，所以，我们可以调用方法 read()将 FSDataInputStream 上的数据读到缓冲区中，并做进一步处理。在下面的代码中，我们先从输入流中读取 1024 大小的数据到缓冲里面，然后将缓冲里的数据写入到输出流 out 里。一直循环，直到从输入流中读到缓冲里的字节长度为 0，表示输入流里的数据已经读取完毕。代码如下：

```
            out = new FileOutputStream(location);
            byte[] ioBuffer = new byte[1024];
```

```
            int readLen = fsin.read(ioBuffer);
            while (-1 != readLen) {
                out.write(ioBuffer, 0, readLen);
                readLen = fsin.read(ioBuffer);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
```

4. 关闭对象实例

```
            finally {
            try {
                if (out != null) {
                    out.close();
                    out = null;
                }
                if (fsin != null) {
                    fsin.close();
                    fsin = null;
                }
                if (fs != null) {
                    fs.close();
                    fs = null;
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

上面的第 3 步和第 4 步骤，也可以只使用下面一个 API 即可完成相同功能：

```
IOUtils.copyBytes(fsin, out, 1024, true);
```

上述方法的 true 参数表示让系统自动关闭输入输出流对象。

4.3.2 写 HDFS 文件内容

下面这个代码是读取一个本地文件，然后将其内容写到 HDFS 上。在创建了一个 org.apache.hadoop.fs.FileSystem 对象之后，就可以使用 create()方法在 HDFS 上创建一个文件（如

果该文件存在，则系统直接覆盖）。create()方法返回一个 FSDataOutputStream 对象，然后就可以使用 IOUtils.copyBytes()方法写数据了。

```
    public   static   FileSystem   uploadToHdfs(String   location,   String
hdfsPath)   {
        InputStream in = null;
        FileSystem fs = null;
        FSDataOutputStream out = null;
        try {
            if (hdfsPath.indexOf("hdfs") > -1) {
                hdfsPath                                                    =
hdfsPath.substring(hdfsPath.lastIndexOf("hdfs"));
            }
            in = new BufferedInputStream(new FileInputStream(location));
            fs       =       FileSystem.get(URI.create(hdfsPath),       (new
Configuration()));
            out = fs.create(new Path(hdfsPath), new Progressable() {
                public void progress() {
                    System.out.print(".");
                }
            });
            IOUtils.copyBytes(in, out, 4096, true);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (out != null) {
                    out.close();
                    out = null;
                }
                if (in != null) {
                    in.close();
                    in = null;
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return fs;
    }
```

4.4 HDFS API 总结

HDFS 的 API 主要在 org.apache.hadoop.fs 包中。本节介绍的类除标明外，默认都在该包下。

4.4.1 Configuration 类

org.apache.hadoop.conf.Configuration 类封装了一个客户端或服务器的配置，用于存取配置参数。系统资源决定了配置的内容。一个资源以 xml 形式的数据表示，由一系列的“键-值”对组成。资源可以用 String 或 Path 命名，String 参数指示 hadoop 在 classpath 中查找该资源；Path 参数指示 hadoop 在本地文件系统中查找该资源。默认情况下，Hadoop 依次从 classpath 中加载 core-default.xml（对于 Hadoop 只读）和 core-site.xml（Hadoop 自己的配置文件，在安装目录的 conf 中），完成初始化配置。

4.4.2 FileSystem 抽象类

这是与 Hadoop 的文件系统交互的接口。可以被实现为一个分布式文件系统，或者一个本地文件系统。使用 HDFS 都要获得 FileSystem 对象，可以像操作一个磁盘一样来操作 HDFS。方法如下：

（1）获得 FileSystem 实例

static FileSystem get(Configuration)：从默认位置 classpath 下读取配置。

static FileSystem get(URI,Configuration)：根据 URI 查找适合的配置文件，若找不到则从默认位置读取。URI 的格式大致为 hdfs://localhost/user/sam/test，这个 test 文件应该为 xml 格式。

（2）读取数据

FSDataInputStream open(Path)：打开指定路径的文件，返回输入流。默认 4KB 的缓冲。

abstract FSDataInputStream open(path,int buffersize)：buffersize 为读取时的缓冲大小。

（3）写入数据

FSDataOutputStream create(Path)：打开指定文件，默认是重写文件。会自动生成所有父目录。有 11 个 create 重载方法，可以指定是否强制覆盖已有文件、文件副本数量、写入文件时的缓冲大小、文件块大小以及文件许可。

public FSDataOutputStream append(Path)：打开已有的文件，在其末尾写入数据。

（4）其他方法

boolean exists(path)：判断源文件是否存在。

boolean mkdirs(Path)：创建目录。

abstract FileStatus getFileStatus(Path)：获取一文件或目录的状态对象。

abstract boolean delete(Path f,boolean recursive)：删除文件，当 recursive 为 ture 时,一个非空目录及其内容都会被删除。如果是一个文件，则 recursive 没用。

boolean deleteOnExit(Path)：标记一个文件，在文件系统关闭时删除。

4.4.3　Path 类

用于指定文件系统中的一个文件或目录。Path String 用“/”隔开目录，如果以“/”开头，则表示为一个绝对路径。一般路径的格式为“hdfs://ip:port/directory/file”。

4.4.4　FSDataInputStream 类

InputStream 的派生类，这是文件输入流，用于读取 HDFS 文件。支持随机访问，可以从流的任意位置读取数据。完全可以当成 InputStream 来进行操作和封装使用。方法如下：

int read(long position,byte[] buffer,int offset,int length)：从 position 处读取 length 字节放入缓冲 buffer 的指定偏离量 offset。返回值是实际读到的字节数。

void readFully(long position,byte[] buffer)，void readFully(long position,byte[] buffer,int offset,int length)：readFully()方法会读出指定位置（也可由 length 指定长度）的数据到 buffer 中，或在只接受 buffer 字节数组的方法中读取“buffer.length”个字节，若已经到文件末，将会抛出 EOFException。

long getPos()：返回当前位置，即距文件开始处的偏移量.

void seek(long desired)：定位到 desired 偏移处，是一个高开销的操作。

4.4.5　FSDataOutputStream 类

OutputStream 的派生类，这是文件输出流，用于写 HDFS 文件。不允许定位，只允许对一个打开的文件顺序写入。方法除 getPos 特有的方法外，继承了 DataOutputStream 的 write 系列方法。

4.4.6　IOUtils 类

org.apache.hadoop.io.IOUtils 类是与 I/O 相关的实用工具类。里面的方法都是静态：

```
    static  void  copyBytes(InputStream  in,OutputStream  out,Configuration
conf)
    static void copyBytes(InputStream in, OutputStream out,Configuration
conf,boolean close)
    static  void  copyBytes(InputStream,OutputStream,int  buffsize,boolean
```

```
close)
    static void copyBytes(InputStream in,OutputStream out,int buffSize)
```

copyBytes 方法把一个流的内容拷贝到另外一个流。close 参数指定了在拷贝结束后是否关闭流，默认为关闭。

static void readFully(InputStream in,byte[] buf, int off,int len)：读数据到 buf 中。

4.4.7 FileStatus 类

用于向客户端显示文件信息，封装了文件系统中文件和目录的元数据，包括文件长度、块大小、副本、修改时间、所有者以及许可信息。

4.4.8 FsShell 类

提供了访问 FileSystem 的命令行，这是带有主函数 main 的类，可以直接运行，如：

```
java FsShell [-ls] [rmr]
```

4.4.9 ChecksumFileSystem 抽象类

为每个源文件创建一个校验文件，在客户端产生和验证校验。

4.4.10 其他 HDFS API 实例

下面我们给出 HDFS API 的更多使用例子，主要说明一下在 Java 程序中如何对 HDFS 里的文件进行创建、删除、查询等操作。

1. 创建文件

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
fs.create(new Path(hdfsPath));
```

create 方法有多种重载，详细情况可参阅 API 文档。

2. 创建目录

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
fs.mkdirs(new Path(hdfsPath));
```

mkdirs 方法有多种重载，详细情况可参阅 API 文档。和上边的 create 方法一样，都会根据

path 建立相应的文件或目录，如果父级目录不存在，则自动创建。如果这并非你所期望的，需要先对路径中的各级目录进行判断。

3. 检查目录或文件是否存在

```
Configuration conf = new Configuration();
FileSystem fs = FileSystem.get(conf);
fs.exists(new Path(hdfsPath));
```

4. 查看文件系统中文件元数据

```
public class getStatus {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        FileStatus stat = fs.getFileStatus(new Path(args[0]));
        System.out.print(stat.getAccessTime()+"
"+stat.getBlockSize()+" "+stat.getGroup()
                +"     "+stat.getLen()+"     "+stat.getModificationTime()+"
"+stat.getOwner()
                +" "+stat.getReplication()+" "+stat.getPermission()
                );
    }
}
```

这个元数据包括文件长度、块大小、备份、修改时间、所有者以及权限信息。FileStatus 有一个 isDir()方法，能够判断是否为目录或是否存在，如果判断是否存在使用 exists 方法比较方便。

5. 查看目录列表

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FileUtil;
import org.apache.hadoop.fs.Path;

public class getPaths {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);
    FileStatus[] statu = fs.listStatus(new Path(args[0]));
    Path [] listPaths=FileUtil.stat2Paths(statu);
```

```
        for(Path p:listPaths){
            System.out.println(p);
        }
    }
    }
```

上面的是 FileSystem 对象的 listStatus()方法，有多个重载，可以传入一个 Path 数组，同时查询多个给定的路径。如果需要查询子目录的路径，需要另写一个函数做递归调用。

6. 删除文件和目录

使用 FileSystem 对象的 delete(Path f,boolean recursive)方法，布尔值设置为 true 时，才会删除一个目录。

7. 通配符操作

以上的一些程序是不适用*、[]等通配符参数的。FileSystem 对象提供有 globStatus()方法可以接受含有通配符的参数。

```
    import org.apache.hadoop.fs.Path;
    import org.apache.hadoop.fs.PathFilter;
    public class pathFilter implements PathFilter{
        private final String regex;
        public pathFilter (String regex){
            this.regex=regex;
        }
        public boolean accept(Path path) {
            return !path.toString().matches(regex);
        }
    }
    public class regxList{
        public static void main(String[] args) throws Exception {
            Configuration conf = new Configuration();
            FileSystem fs = FileSystem.get(conf);
            FileStatus[]  statu  =  fs.globStatus(new  Path(args[0]),  new
pathFilter ("^2016"));
            Path [] listPaths=FileUtil.stat2Paths(statu);
            for(Path p:listPaths){
                System.out.println(p);
            }
        }
    }
```

上面的 PathFilter 是用来过滤通配符不需要匹配的内容。

8. 验证是否是文件

```
boolean isFile = fs.isFile(inputPath);
```

9. 重命名

```
boolean renamed=fs.rename(inputPath,new Path("新名字"));
```

4.4.11 综合实例

下面我们编写一个稍微复杂的程序：在指定文件目录下的所有文件中，检索某一特定字符串所出现的行，将这些行的内容输出到本地文件系统的输出文件夹中。这个程序假定只有第一层目录下的文件才有效，而且，假定文件都是文本文件。为了防止单个的输出文件过大，这里还加了一个文件最大行数限制，当文件行数达到最大值时，便关闭此文件，创建另外的文件继续保存。保存的结果文件名为 1，2，3，4，…，以此类推。因为这个程序可以用来分析 MapReduce 的结果，所以称为 ResultFilter。程序 ResultFilter 接收 4 个命令行输入参数，参数含义如下：

- <dfs path>：HDFS 上的路径
- <local path>：本地路径
- <match str>：待查找的字符串
- <single file lines>：结果的每个文件的行数

程序 ResultFilter 如下：

```
import java.util.Scanner;
import java.io.IOException;
import java.io.File;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
public class resultFilter
{
    public static void main(String[] args) throws IOException {
        Configuration conf = new Configuration();
        // hdfs 和 local 分别对应 HDFS 实例和本地文件系统实例
        FileSystem hdfs = FileSystem.get(conf);
        FileSystem local = FileSystem.getLocal(conf);
```

```
        Path inputDir, localFile;
        FileStatus[] inputFiles;
        FSDataOutputStream out = null;
        FSDataInputStream in = null;
        Scanner scan;
        String str;
        byte[] buf;
        int singleFileLines;
        int numLines, numFiles, i;
        if(args.length!=4)
        {
            // 输入参数数量不够,提示参数格式后终止程序执行
            System.err.println("usage  resultFilter  <dfs  path><local
path>" +
            " <match str><single file lines>");
            return;
        }
        inputDir = new Path(args[0]);
        singleFileLines = Integer.parseInt(args[3]);
        try {
            inputFiles = hdfs.listStatus(inputDir); // 获得目录信息
            numLines = 0;
            numFiles = 1;    // 输出文件从1开始编号
            localFile = new Path(args[1]);
            if(local.exists(localFile))  // 若目标路径存在,则删除之
                local.delete(localFile, true);
            for  (i = 0; i<inputFiles.length; i++) {
              if(inputFiles[i].isDir() == true) // 忽略子目录
                        continue;
             System.out.println(inputFiles[i].getPath().getName());
              in = hdfs.open(inputFiles[i].getPath());
              scan = new Scanner(in);
              while (scan.hasNext()) {
                 str = scan.nextLine();
                 if(str.indexOf(args[2])==-1)
                     continue;  // 如果该行没有 match 字符串,则忽略之
                 numLines++;
                 if(numLines == 1)   // 如果是1,说明需要新建文件了
                 {
```

```
                    localFile=new Path(args[1]+File.separator+numFiles);
                    out = local.create(localFile); // 创建文件
                    numFiles++;
                }
                buf = (str+"\n").getBytes();
                out.write(buf, 0, buf.length); // 将字符串写入输出流
                if(numLines == singleFileLines)//如果已满足相应行数,关闭文件
                {
                    out.close();
                    numLines = 0;   // 行数变为0,重新统计
                }
            }// end of while
            scan.close();
            in.close();
           }// end of for
           if(out != null)
               out.close();
           } // end of try
           catch (IOException e) {
               e.printStackTrace();
           }
       }// end of main
   }// end of resultFilter
```

上述程序的逻辑很简单，获取该目录下所有文件的信息，对每一个文件，打开文件、循环读取数据、写入目标位置，然后关闭文件，最后关闭输出文件。

运行命令如下：

```
    hadoop jar resultFilter.jar resultFilter <dfs path> <local path>
<match str> <single file lines>
```

4.5 HDFS 文件格式

本节介绍 Hadoop 目前已有的几种文件格式，分析其特点、开销及使用场景。希望加深读者对 Hadoop 文件格式及其性能影响因素的理解。

4.5.1 SequenceFile

SequenceFile 是 Hadoop API 提供的一种二进制文件，它将数据以<key,value>的形式序列化到文件中，如图 4-11 所示。这种二进制文件在内部使用 Hadoop 标准的 Writable 接口实现序列化和反序列化。它与 Hadoop API 中的 MapFile 是互相兼容的。

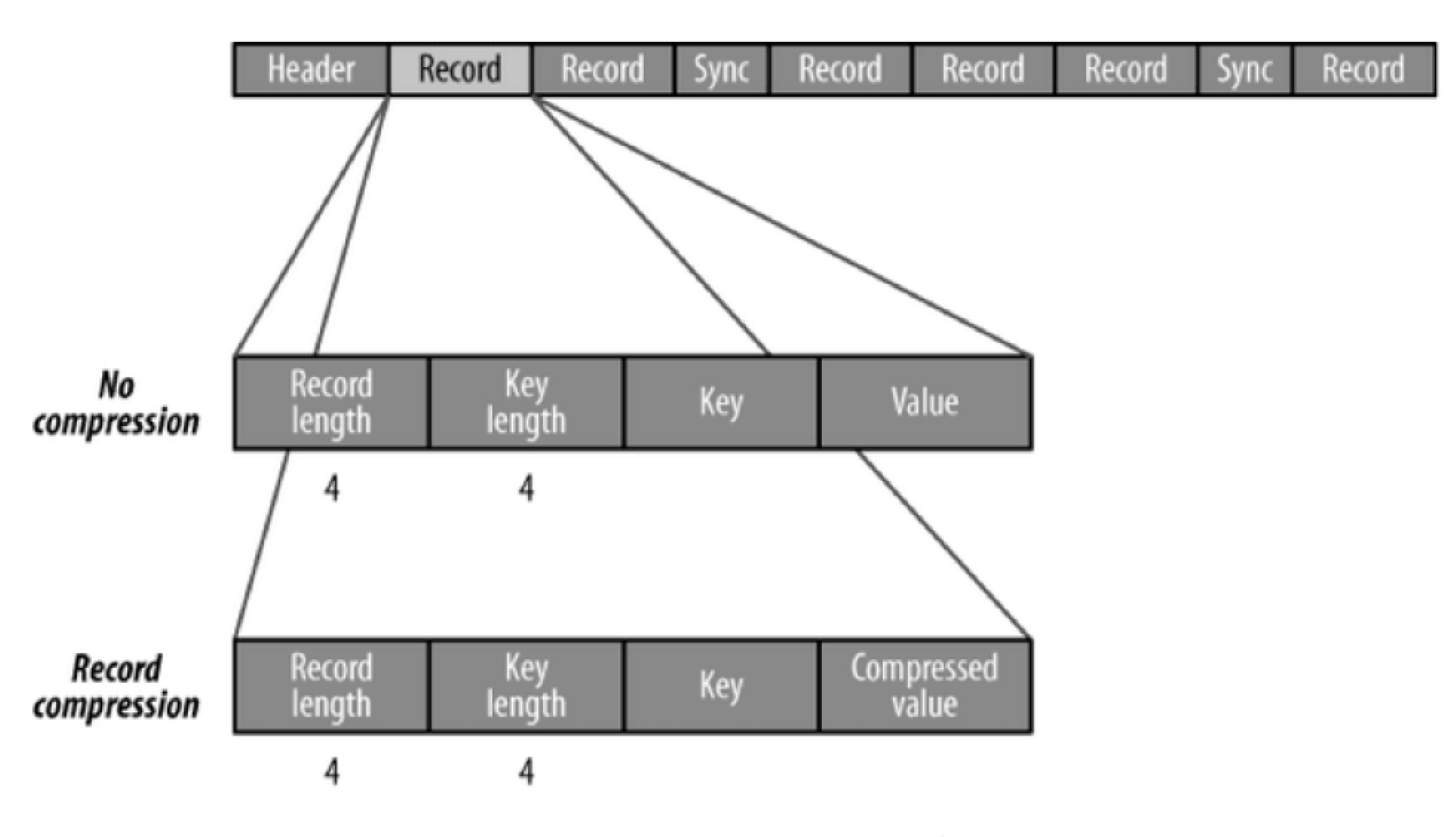


图 4-11　Sequencefile 文件结构

4.5.2 TextFile（文本格式）

上面提到的 SequenceFile 是二进制格式。文本格式的数据也是 Hadoop 中经常碰到的。如文本文件、XML 和 JSON。文本格式除了会占用更多磁盘资源外，对它的解析开销一般也会比二进制格式高几十倍以上，尤其是 XML 和 JSON，它们的解析开销比文本文件还要大，因此不建议在生产系统中使用这些格式进行储存。 如果需要输出这些格式，可在客户端做相应的转换操作。文本格式经常会用于日志收集、数据库导入等。另外，文本格式的一个缺点是它不具备类型和模式，比如销售额这类数值数据或者日期时间类型的数据，如果使用文本格式保存，由于它们本身的字符串类型的长短不一，或者含有负数，有时需要将它们预处理成含有模式的二进制格式，这又导致了不必要的预处理步骤的开销和储存资源的浪费。

4.5.3 RCFile

TextFile 和 SequenceFile 的存储格式都是基于行存储的，RCFile（Record Columnar File）是基于行列混合的思想。如图 4-12 所示，先按行把数据划分成 N 个 row group，在 row group 中对每个列分别进行存储。当查询过程中，针对它并不关心的列时，它会在 IO 上跳过这些列。该结构强调的是：

- RCFile 存储的表是水平划分的，分为多个行组，每个行组再被垂直划分，以便每列单独存储；
- RCFile 在每个行组中利用一个列维度的数据压缩，并提供一种 Lazy 解压（decompression）技术。这在查询执行时可避免不必要的列解压；
- RCFile 支持弹性的行组大小，行组大小需要权衡数据压缩性能和查询性能两方面。

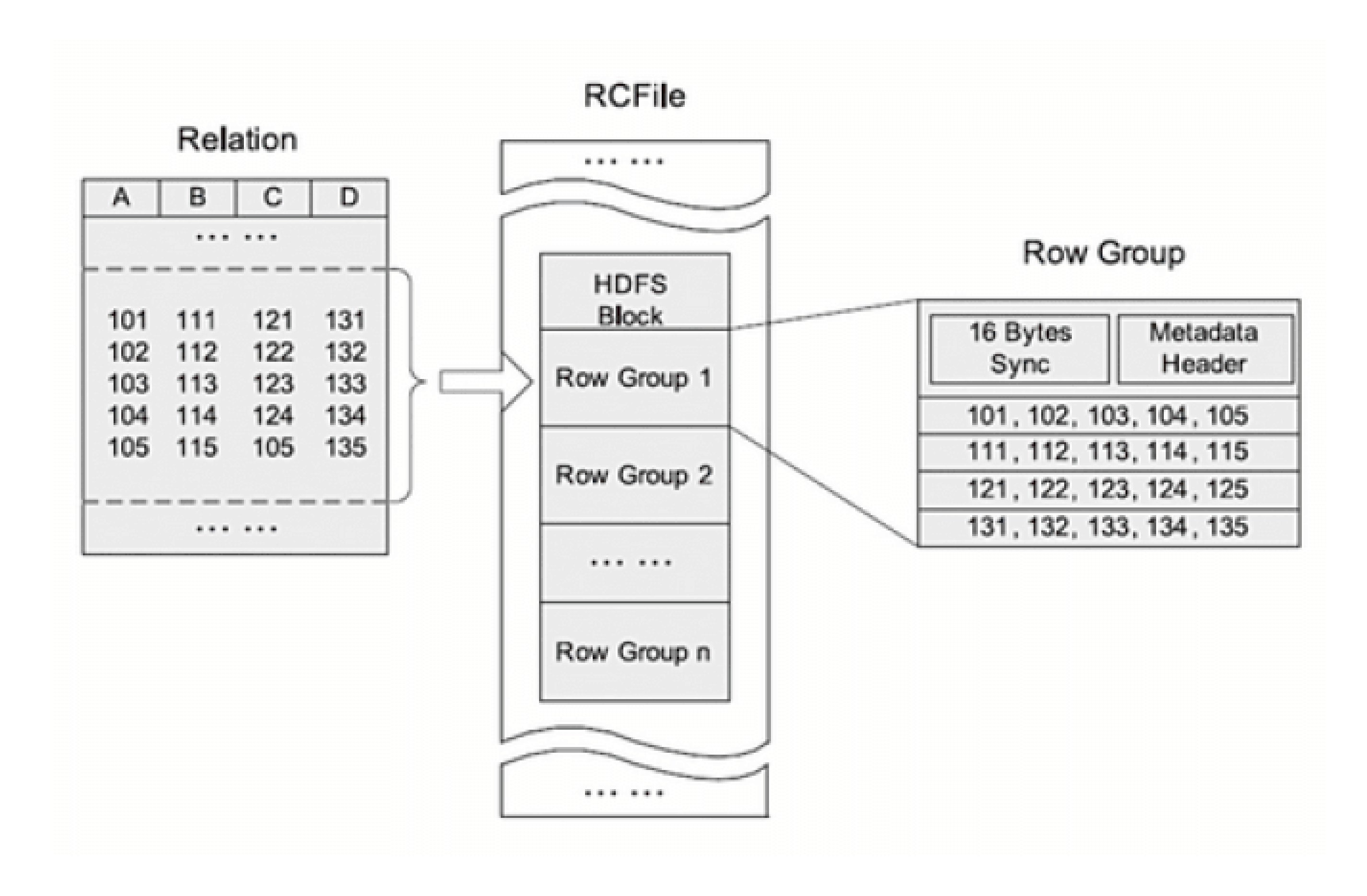


图 4-12　RCFile 文件结构

RCFile 存储结构遵循的是“先水平划分，再垂直划分”的设计理念，这个想法来源于 PAX。它结合了行存储和列存储的优点：首先，RCFile 保证同一行的数据位于同一节点，因此元组重构的开销很低；其次，像列存储一样，RCFile 能够利用列维度的数据压缩，并且能跳过不必要的列读取。

在图 4-12 中，RCFile 基于 HDFS 架构，表格占用多个 HDFS 块。每个 HDFS 块中，RCFile 以行组为基本单位来组织记录。也就是说，存储在一个 HDFS 块中的所有记录被划分为多个行组。对于一张表，所有行组大小都相同。一个 HDFS 块会有一个或多个行组。一个行组包括三个部分。

- 第一部分是行组头部的同步标识，主要用于分隔 HDFS 块中的两个连续行组；
- 第二部分是行组的元数据头部，用于存储行组单元的信息，包括行组中的记录数、每个列的字节数、列中每个域的字节数；
- 第三部分是表格数据段，即实际的列存储数据。在该部分中，同一列的所有域顺序存储。从图上可以看出，首先存储了列 A 的所有域，然后存储列 B 的所有域等。

RCFile 的每个行组中，元数据头部和表格数据段分别进行压缩。对于所有元数据头部，RCFile 使用 RLE（Run Length Encoding）算法来压缩数据。由于同一列中所有域的长度值都顺序存储在该部分，RLE 算法能够找到重复值的长序列，尤其对于固定的域长度。表格数据段不会作为整个单元来压缩；相反每个列被独立压缩，使用 Gzip 压缩算法。RCFile 使用 Gzip 压缩算法是为了获得较好的压缩比，而不使用 RLE 算法的原因在于此时列数据非排序。此外，由于 Lazy 压缩策略，当处理一个行组时，RCFile 不需要解压所有列。因此，相对较高的 Gzip 解压

开销可以减少。

RCFile 不支持任意方式的数据写操作，仅提供一种追加接口，这是因为底层的 HDFS 当前仅仅支持数据追加写文件尾部。RCFile 提供两个参数来控制在写到磁盘之前，内存中缓存多少个记录。一个参数是记录数的限制，另一个是内存缓存的大小限制。

在 MapReduce 框架中，mapper 将顺序处理 HDFS 块中的每个行组。当处理一个行组时，RCFile 无须全部读取行组的全部内容到内存。它仅仅读元数据头部和给定查询需要的列。因此，它可以跳过不必要的列以获得列存储的 I/O 优势。例如，表 tbl(c1, c2, c3, c4)有 4 个列，做一次查询“SELECT c1 FROM tbl WHERE c4=1”，对每个行组，RCFile 仅仅读取 c1 和 c4 列的内容。在元数据头部和需要的列数据加载到内存中后，它们需要解压。元数据头部总会解压并在内存中存放，直到 RCFile 处理下一个行组。然而，RCFile 不会解压所有加载的列，相反，它使用一种 Lazy 解压技术。Lazy 解压意味着列将不会在内存解压，直到 RCFile 决定列中数据真正对查询执行有用。由于查询使用各种 WHERE 条件，Lazy 解压非常有用。如果一个 WHERE 条件不能被行组中的所有记录满足，那么 RCFile 将不会解压 WHERE 条件中不满足的列。例如，在上述查询中，所有行组中的列 c4 都解压了。然而，对于一个行组，如果列 c4 中没有值为 1 的域，那么就无须解压列 c1。

I/O 性能是 RCFile 关注的重点，因此 RCFile 需要行组够大并且大小可变。行组够大的话，数据压缩效率会比行组小时更有效。尽管行组变大有助于减少表格的存储规模，但是可能会损害数据的读性能，因为这样减少了 Lazy 解压带来的性能提升。而且行组变大会占用更多的内存，这会影响并发执行的其他 MapReduce 作业。考虑到存储空间和查询效率两个方面，默认的行组大小为 4MB，当然也允许用户自行选择参数进行配置。

RCFile 存储结构广泛应用于 Hive 中。首先，RCFile 具备相当于行存储的数据加载速度和负载适应能力；其次，RCFile 的读优化可以在扫描表格时避免不必要的列读取，它比其他结构拥有更好的性能；再次，RCFile 使用列维度的压缩，因此能够有效提升存储空间利用率。为了提高存储空间利用率，Facebook 各产品线应用产生的数据从 2010 年起均采用 RCFile 结构存储，按行存储 （SequenceFile/TextFile）结构保存的数据集也转存为 RCFile 格式。此外，在 Pig 数据分析系统中也集成了 RCFile。

4.5.4 Avro

Avro 是一种用于支持数据密集型的二进制文件格式。它的文件格式更为紧凑，若要读取大量数据时，Avro 能够提供更好的序列化和反序列化性能。并且 Avro 数据文件天生是带 Schema 定义的，所以它不需要开发者在 API 级别实现自己的 Writable 对象。多个 Hadoop 子项目都支持 Avro 数据格式，如 Pig 、Hive、Flume、Sqoop 和 Hcatalog。

第 5 章

大数据存储：数据库

从历史上看，数据处理主要是通过数据库技术来实现的。大多数数据都有定义良好的结构，数据集不大，可以通过关系数据库存储和查询。然而，在大数据的世界里，基于表的传统的关系型数据库（RDBMS，比如：MySQL、Oracle、DB2 UDB、SQL Server 等）并不适合，这是因为单个表在数据量变得巨大时就显得力不从心，而且 RDBMS 在横向扩展上非常弱。HBase 是 Apache Hadoop 的数据库，是一个非关系型的 NoSQL 数据库。它能够对大型数据提供随机、实时的读写访问，HBase 就是一个能很好地存储并处理海量数据的数据库。Facebook 的 Messaging 平台就是基于 HBase 开发的。在本章，我们首先解释一下什么是 NoSQL，然后重点介绍 HBase 数据库。

5.1 NoSQL

NoSQL 是 Not only SQL 的缩写，泛指非关系型数据库。与 RDBMS 相比，NoSQL 不使用 SQL 作为查询语言，其表没有固定的结构，具有水平扩展的特性，非常容易支撑 TB 乃至 PB 的数据量。下面列出了 NoSQL 的几个优点：

- 易扩展

NoSQL 数据库种类繁多，但是一个共同的特点都是去掉关系数据库的关系型特性。数据之间无关系，这样就非常容易扩展。也无形之中，在架构的层面上给用户带来了可扩展的能力。

- 大数据量，高性能

NoSQL 数据库都具有非常高的读写性能，尤其在大数据量下表现优秀。这得益于它的无关系性，数据库的结构简单。一般 MySQL 使用 Query Cache，每次表一更新 Cache 就失效，是一种大粒度的 Cache。而 NoSQL 的 Cache 是记录级的，是一种细粒度的 Cache，所以 NoSQL 在这个层面上来说性能就高很多了。

- 灵活的数据模型

NoSQL 无须事先为要存储的数据建立字段，随时可以存储自定义的数据格式。而在关系数据库里，增删字段是一件非常麻烦的事情。

- 高可用

NoSQL 在不太影响性能的情况下，就可以方便地实现高可用的架构。比如：HBase 模型就是通过复制模型也能实现高可用。

对于数据库产品而言，底层存储结构直接决定了数据库的特性和使用场景。RDBMS 使用 B 树和 B+树作为存储结构，而 NoSQL 之一的 HBase 的底层存储结构使用 LSM 树作为存储结构。B+树的特点是能够保持数据稳定有序，通过最大化每个内部节点中的子节点的数量来减少树的高度，从而增加效率。LSM 树通过使用某种算法对索引变更进行延迟及批量处理，并通过一种类似归并排序的方式，联合使用一个基于内存的组件和一个或多个磁盘组件。在处理过程中，所有的索引值对于所有的查询来说，都可以通过内存组件或者某个磁盘组件进行访问。与 B+树相比，这就大大减少了磁盘磁臂的移动次数，提高了读写性能。对于磁盘而言，LSM 树属于传输型，而 RDBMS 是属于寻道型。在大数据的情况下，计算瓶颈主要在磁盘的数据传输上，这也是为什么 LSM 被用于大数据场景。

NoSQL 数据库有多种，分为行存储和列存储。不同的 NoSQL 数据库适用不同的场景，一部分在查询数据（select）时性能更好，有些是在插入或者更新上性能更好。具体的数据库选型依赖于你的具体需求（例如，你的应用程序的数据库读写比）。压缩率、缓冲池、超时的大小和缓存，对于不同的 NoSQL 数据库来说配置都是不同的，同时对数据库性能的影响也是不一样的。并非所有的 NoSQL 数据库都内置了支持连接、排序、汇总、过滤器、索引等特性。如果有需要，还是建议使用具有这些内置功能的数据库。NoSQL 数据库内置了压缩、编解码器和数据移植工具。

5.2 HBase 管理

HBase 是一个开源、分布式的、高性能的、可扩展的、面向列的 NoSQL 数据库。它是 Apache Hadoop 生态系统中的重要一员，主要用于海量结构化数据存储。当你需要对大数据进行实时的、随机的存储和访问，你就可以使用 HBase。HBase 源于 Google 的一篇论文《bigtable：一个结构化数据的分布式存储系统》，HBase 是 Google Bigtable 的开源实现，它利用 Hadoop HDFS 作为其文件存储系统，利用 Hadoop MapReduce 来处理 HBase 中的海量数据，利用 Zookeeper 作为协同服务。HBase 使用 Key-Value 存储，HDFS 为 HBase 提供了高可靠的底层存储支持，而 MapReduce 为 HBase 提供了高性能的计算能力。HBase 弥补了 Hadoop 只能离线批处理的不足，为 Hadoop 提供了实时处理数据的能力。HBase 的整个项目使用 Java 语言实现。

HBase 是一个在 HDFS 上开发的面向列的分布式数据库。从逻辑上讲，HBase 将数据按照表、行和列进行存储。与 HDFS 一样，HBase 主要依靠横向扩展，通过不断增加廉价的商用服务器，来增加计算和存储能力。HBase 表的特点如下：

- 容量大：一个表可以有数十亿行，上百万列。当关系型数据库（如：ORACLE）的单个表的记录在亿级时，则查询和写入的性能都会呈现指数级下降，而 HBase 对于单表存储百亿或更多的数据都没有性能问题。
- 无固定模式（表结构不固定）：每行都有一个可排序的主键和任意多的列，列可以根据需要动态地增加，同一张表中不同的行可以有截然不同的列。
- 面向列：面向列（簇）的存储和权限控制，支持列（簇）独立检索。RDBMS 是按行存储的，在数据量大的时候，RDBMS 依赖索引来提高查询速度，而建立索引和更新索引需要大量的时间和空间。对于 HBase 而言，因为数据是按照列存储，每一列都单独存放，所以数据即索引，在查询时可以只访问所涉及的列的数据，大大降低了系统的 I/O。
- 稀疏性：空（null）列并不占用存储空间，表可以设计的非常稀疏。
- 数据多版本：每个单元中的数据可以有多个版本，默认情况下版本号自动分配，是单元格插入时的时间戳。
- 数据类型单一：HBase 中的数据都是字符串，没有类型。
- 高性能：针对 Rowkey 的查询能够达到毫秒级别。

5.2.1　HBase 表结构

类似 RDBMS，HBase 也以表的形式存储数据，如图 5-1 所示。表也由行和列组成。但是，与 RDBMS 不同的是，HBase 表的每一行都有唯一的行键（row key），原来 RDBMS 的列被划分为若干个列簇（column family），每一行有相同的列簇，列簇将一列或多列组合在一起，HBase 的列必须属于某一个列簇。相同列簇可以有不同的列，每个列可以有多个版本的数据，指定版本获取数据。HBase 允许用户存储大量的信息到一个表中，而 RDBMS 的大量信息则可能被分到多个表上存储。

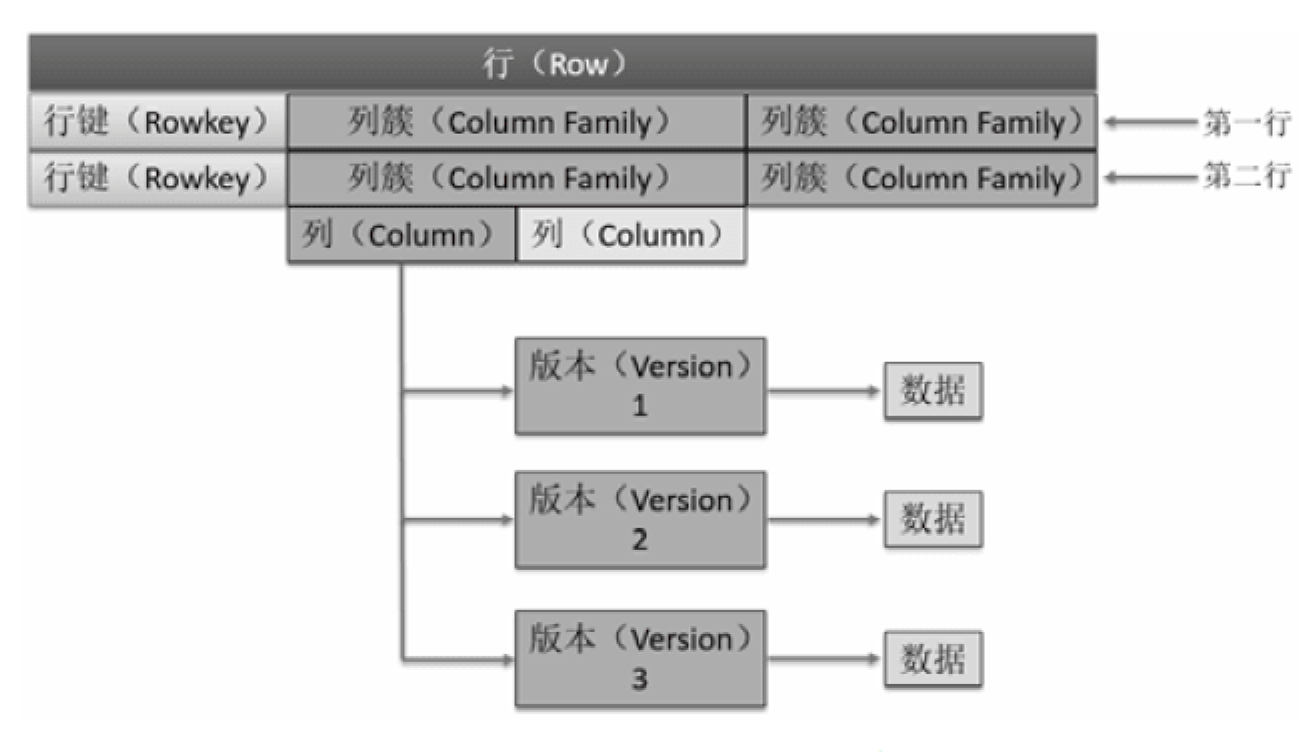


图 5-1　HBase 表结构

在表 5-1 中，key1、key2 是两条记录的唯一的行键（row key）值。列簇 1、列簇 2 和列簇 3 是三个列簇，每个列簇下又包括几列。比如列簇 1 包括两列，名字是列 1 和列 2，“t1:abc”和“t2:gdxdf”是由 key1 和“列簇 1-列 1”唯一确定的一个单元 cell。这个 cell 中有两个数据：abc 和 gdxdf。两个值的时间戳不一样，分别是 t1 和 t2, HBase 会返回最新时间的值给请求者。

表 5-1　HBase 表结构示例

行键（Row Key）	列簇 1		列簇 2			列簇 3
	列 1	列 2	列 1	列 2	列 3	列 1
key1	t1:abc t2:gdxdf					
key2						

HBase 中的几个术语的具体含义如下：

（1）行键（row key）

row key 是用来唯一确定一行的标识，不同的行键代表不同的行，它是检索记录的主键，必须在设计上保证其唯一性。访问 HBase table 中的行，只有三种方式：

- 通过单个 row key 访问
- 通过 row key 的 range
- 全表扫描

行键（row key）可以是任意字符串，它的最大长度是 64KB，实际应用中长度一般为 10~100 字节。在 HBase 内部，row key 被保存为字节数组。存储时，数据按照 row key 的字典序（byte order）排序存储。设计 key 时，要充分考虑排序存储这个特性，将经常一起读取的行存储放到一起（位置相关性）。比如：字典序对 int 排序的结果是：1,10,100,11,12,13,14,15,16,17,18,19,2,20,21,…,9,91。如果要保持整形的自然序，行键必须用 0 作左填充。另外，行的一次读写是原子操作，不论一次读写多少列。

为了高效检索数据，我们应该仔细设计 row key 以获得最高的查询性能。row key 应该尽量均匀分布。因为 HBase 的行键是有序排列的，所以，我们应该避免单调递增行键。否则，写入数据时，就会集中对某一个 Region 进行写入操作，这时候所有的负载都在同一台机器上。这对全表扫描的读操作也是如此。对于行键的长度，既要满足语义，又要尽量缩短以减少存储空间。

（2）列簇（column family）

HBase 表中的每个列，都归属于某个列簇。列簇是表的 schema 的一部分（而列不是），必须在使用表之前定义。每个表必须至少要有一个列簇。列名都以列簇作为前缀，并用冒号分隔开。例如 courses:history,courses:math 都属于 courses 这个列簇。访问控制、磁盘和内存的使用统计都

是在列簇上进行的，所以应该将经常一起查询的列放在一个列簇中，以提高查询的效率。比如，我们有一个会员表，这个表上包含了两部分信息：一部分是会员的基本信息（地址，年龄，名字，电话，住址等），这些信息基本不改动；另外一部分是会员的行为信息（这个数据的读写频率高）。这样的话，我们可以把这两部分信息通过两个列簇分开。

新的列簇可以随后按需动态加入（修改列簇前要先停用表）。但是，我们不推荐有太多的列簇，因为跨列簇的访问是非常低效的。还有，列簇的名字尽量短小，这样可以节省存储空间，提高查询的速度。在实际应用中，列簇上的控制权限能帮助我们管理不同类型的应用：我们允许一些应用可以添加新的基本数据，一些应用则只允许浏览数据（甚至可能因为隐私的原因，不能浏览所有数据）。

与 RDBMS 不同的是，HBase 的表没有列定义，没有数据类型，这也是 HBase 被称为无模式数据库的原因。

（3）单元（cell）

HBase 中通过行和列所确定的一个存储单元，称为 cell，就是传统关系型数据库上的列值。它是版本化的，它是由{row key, column(=<family> + <label>), version} 唯一确定的单元。HBase 没有数据类型，cell 中的数据全部是字节数组，以二进制形式存储。在默认情况下，HBase 的每个单元只维护三个时间版本。如果需要不同的版本数，可以在创建表时指定。单元还有生存时间（Time To Live，TTL），如果过期，则系统会将其删除，也可以在建表时设置 TTL。

（4）时间戳（timestamp）

每个 cell 都保存着同一份数据的多个版本。版本通过时间戳来索引。时间戳的类型是 64 位整型。时间戳可以由 HBase 在数据写入时自动赋值，此时时间戳是精确到毫秒的当前系统时间。时间戳也可以由客户显式赋值。如果应用程序要避免数据版本冲突，就必须自己生成具有唯一性的时间戳。每个 cell 中，不同版本的数据按照时间倒序排序，即最新的数据排在最前面。为了避免数据存在过多版本造成的管理负担，包括存储和索引，HBase 提供了两种数据版本回收方式。一是保存数据的最后 n 个版本，二是保存最近一段时间内的版本（比如最近 7 天）。用户可以针对每个列簇进行设置。

如图 5-2 所示，随着一个表的记录增多而不断变大，会自动分裂成多份，成为 Regions（关于 Region 的更多信息，请参见下节内容）。一个 region 由[startkey，endkey]表示，不同 region 会被 Master 分配给相应的 RegionServer 进行管理。

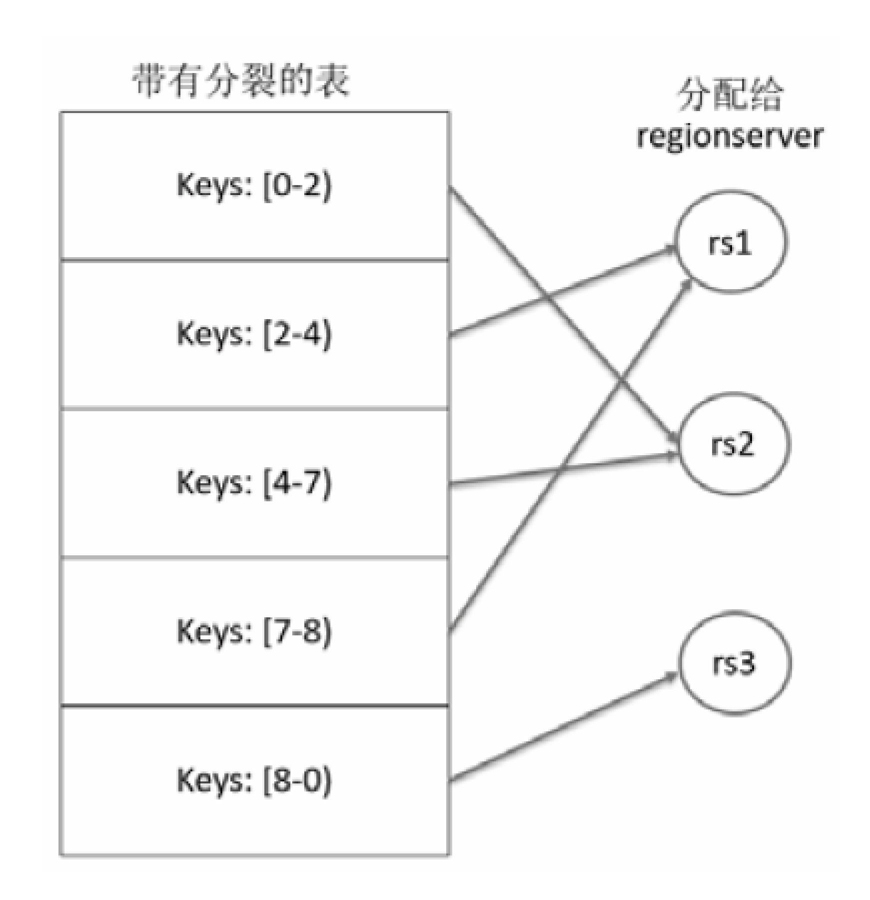


图 5-2 表分裂

最后，我们在表 5-2 中总结了 HBase 与 RDBMS 的区别。

表 5-2 HBase 与 RDBMS 的区别

属性	HBase	RDBMS
数据类型	只有字符串	丰富的数据类型
数据操作	简单的增删改查，本身不支持连接	各种各样的操作
存储模式	基于列式存储	基于表格结构和行式存储
数据保护	更新后仍然可以保留旧版本	替换
可伸缩性	容易增加新节点	复杂

5.2.2 HBase 系统架构

如图 5-3 所示，HBase 包含了客户端应用 client、主节点 HMaster 和 Region 节点 HRegionServer。

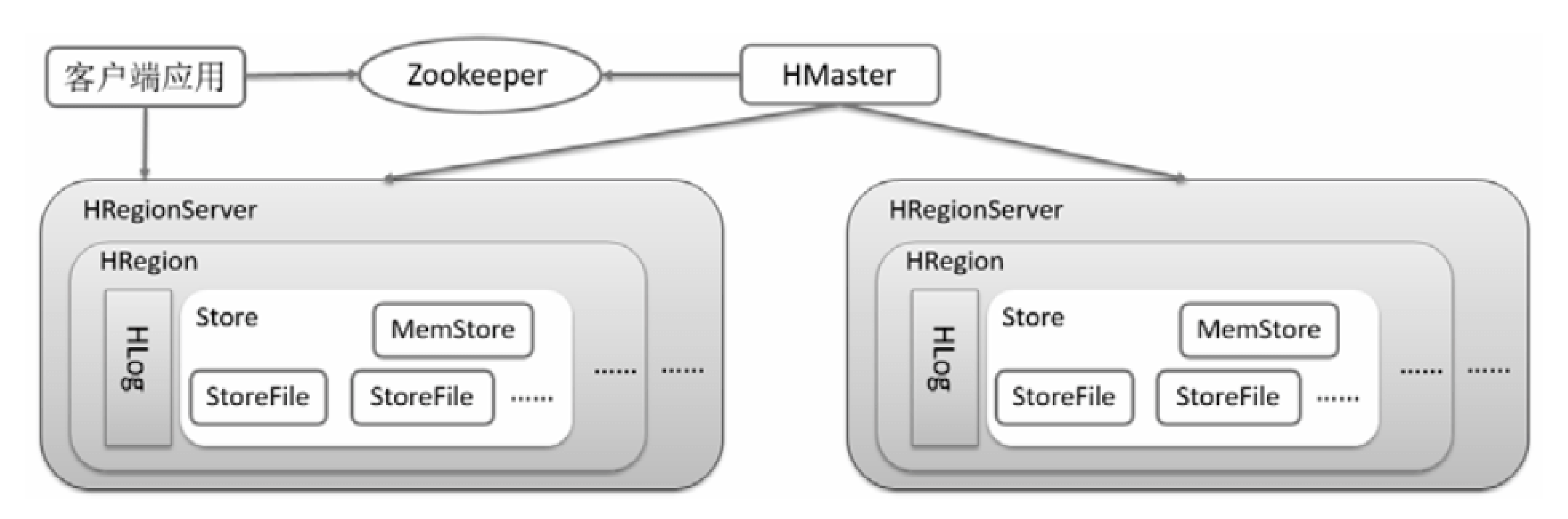


图 5-3 HBase 架构

HBase Client 使用 HBase 的 RPC 机制与 HMaster 和 HRegionServer 进行通信，对于管理类操作，Client 与 HMaster 进行 RPC；对于数据读写类操作，Client 与 HRegionServer 进行 RPC。

HMaster 在功能上主要负责 Table 和 Region 的管理工作：

- 管理用户对 Table 的增、删、改、查操作;
- 管理 HRegionServer 的负载均衡，调整 Region 分布;
- 在 Region Split 后，负责新 Region 的分配;
- 在 HRegionServer 停机后，负责失效 HRegionServer 上的 Regions 迁移。

HMaster 没有单点问题，HBase 中可以启动多个 HMaster，通过 Zookeeper 的 Master Election 机制保证总有一个 Master 运行。HRegionServer 主要负责响应用户 I/O 请求，向 HDFS 文件系统中读写数据，它是 HBase 中最核心的模块。在 HDFS 中可以看到每个表的表名作为独立的目录结构。如图 5-3 所示，HRegionServer 内部管理了一系列 HRegion 对象，每个 HRegion 对应了 Table 中的一个 Region ，HRegion 中由多个 HStore 组成。每个 HStore 对应了 Table 中的一个 Column Family 的存储，可以看出每个 Column Family 其实就是一个集中的存储单元，因此最好将具备共同 I/O 特性的列放在一个 Column Family 中，这样最高效。HRegionServer 也会把自己注册到 Zookeeper 中，使得 HMaster 可以随时感知到各个 HRegionServer 的健康状态。

HStore 存储是 HBase 存储的核心，由两部分组成：一部分是 MemStore，一部分是 StoreFiles。MemStore 是 Sorted Memory Buffer，用户写入的数据首先会放入 MemStore，当 MemStore 满了以后会 Flush 成一个 StoreFile（底层实现是 HFile），当 StoreFile 文件数量增长到一定阈值，会触发 Compact 合并操作，将多个 StoreFiles 合并成一个 StoreFile，合并过程中会进行版本合并和数据删除。从中可以看出，HBase 其实只有增加数据，所有的更新和删除操作都是在后续的 compact 过程中进行的，这使得用户的写操作只要进入内存中就可以立即返回，保证了 HBase I/O 的高性能。当 StoreFiles Compact 后，会逐步形成越来越大的 StoreFile。当单个 StoreFile 大小超过一定阈值后，会触发 Split 操作，同时把当前 Region Split 成两个 Region，父 Region 会下线，新 Split 出的两个孩子 Region 会被 HMaster 分配到相应的 HRegionServer 上，使得原先一个 Region 的压力得以分流到两个 Region 上。

在理解了上述 HStore 的基本原理后，还必须了解一下 HLog 的功能，因为上述的 HStore 在系统正常工作的前提下是没有问题的，但是在分布式系统环境中，无法避免系统出错或者宕机，因此一旦 HRegionServer 意外退出，MemStore 中的内存数据将会丢失，这就需要引入 HLog 了。每个 HRegionServer 中都有一个 HLog 对象，HLog 是一个实现 Write Ahead Log 的类，在每次用户操作写入 MemStore 的同时，也会写一份数据到 HLog 文件中，HLog 文件定期会滚动出新的，并删除旧的文件（已持久化到 StoreFile 中的数据）。当 HRegionServer 意外终止后，HMaster 会通过 Zookeeper 感知到，HMaster 首先会处理遗留的 HLog 文件，将其中不同 Region 的 Log 数据进行拆分，分别放到相应 region 的目录下，然后再将失效的 region 重新分配。领取到这些 region 的 HRegionServer 在 Load Region 的过程中，会发现有历史 HLog 需要处理，因此会 Replay HLog 中的数据到 MemStore 中，然后 flush 到 StoreFiles，完成数据恢复。

上面的系统架构也再次说明了，传统关系型数据库（RDBMS）是一行一行存储结构化数据，而 HBase 是一个面向列的分布式数据库，是一列一列存储结构化数据，这可以支持高并发读写数据请求，从而实现数据库的横向扩展。

HBase 在 HDFS 上有一个可配置的根目录，默认设置为/hbase。通过配置文件 hbase-site.xml 可以设置路径。在创建表并导入部分数据之后，可以在 HDFS 上的 HBase 根目录下看到 HBase 的文件。其中一类是位于表目录下面的文件。每个表都有它自己的目录。每个表目录包含一个名为.tableinfo 的顶层文件，该文件保存了该表的 HTableDescriptor 序列化后的内容，包含了元数据信息。在每个表目录内，针对每个列簇都会有一个单独的目录，这个目录名称包含了 Region 名称的部分信息。当一个 Region 内的存储文件大于 hbase.hregion.max.fielsize 时，该 Region 就需要 split 为两个。该过程非常快，因为系统只是为新 Region 创建两个引用文件，每个只持有原来 Region 一半的内容。HBase 在 StoreFile 内使用一种称为 HFile 的文件存储格式来存储数据。文件是变长的，定长的块只有 file info 和 trailer 两个部分，而 trailer 中包含了指向其他数据块的指针（注意，这是文件内的数据块大小，默认为 64KB，这不是 HDFS 的数据块大小）。文件的每个数据库包含了一系列序列化的 KeyValue 对象和 Magic 头。查询数据的 Get 方法就是通过 Key 查找 Value。

5.2.3 启动并操作 HBase 数据库

下面我们启动并操作 HBase 数据库。

步骤 01 进入 Ambari，找到 HBase 组件，选择启动。如图 5-4 所示。

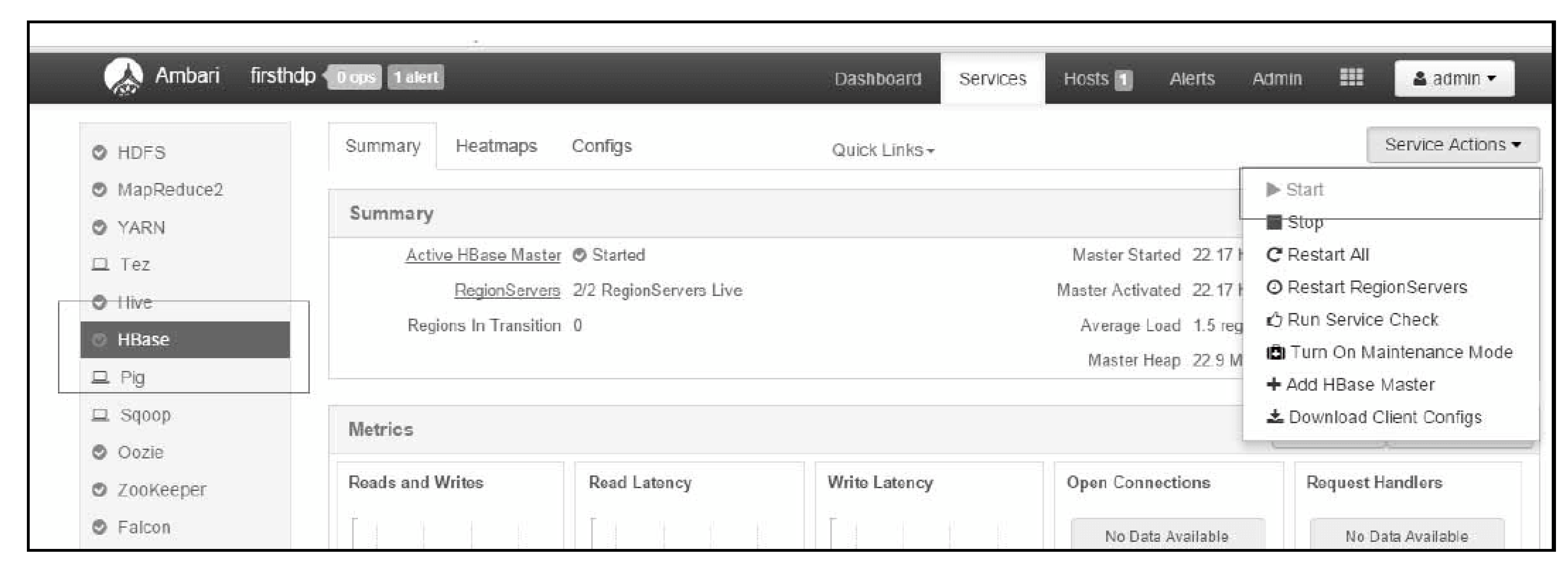


图 5-4 启动 HBase

步骤 02 如果 HBase 出现如图 5-5 所示绿色的对号小图标，即 started，则显示 hbase 启动成功。

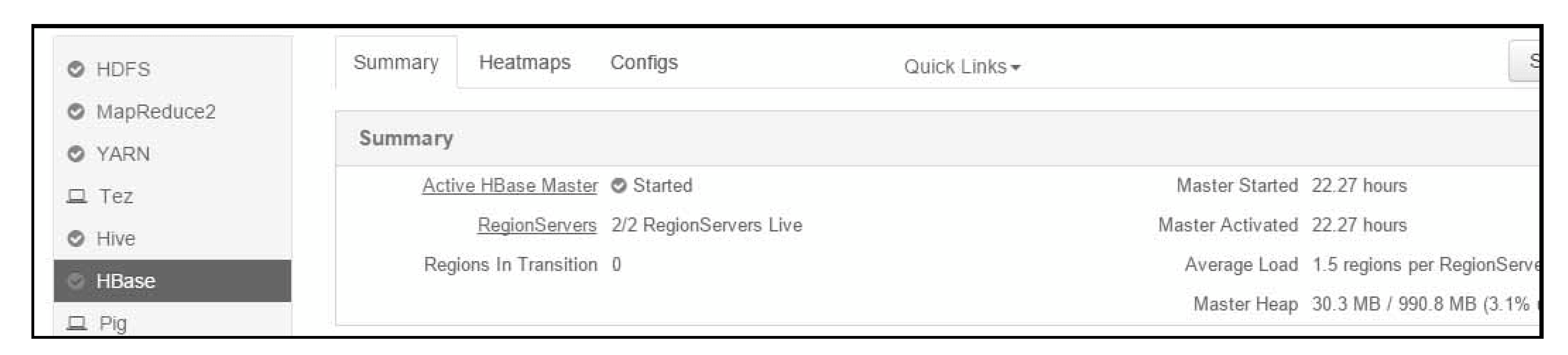


图 5-5 HBase 状态

步骤 03　Ambari 提供 HBase 详细信息的图形化界面，并提供监测相关的小插件，插件可以自行添加删除，如图 5-6 所示。

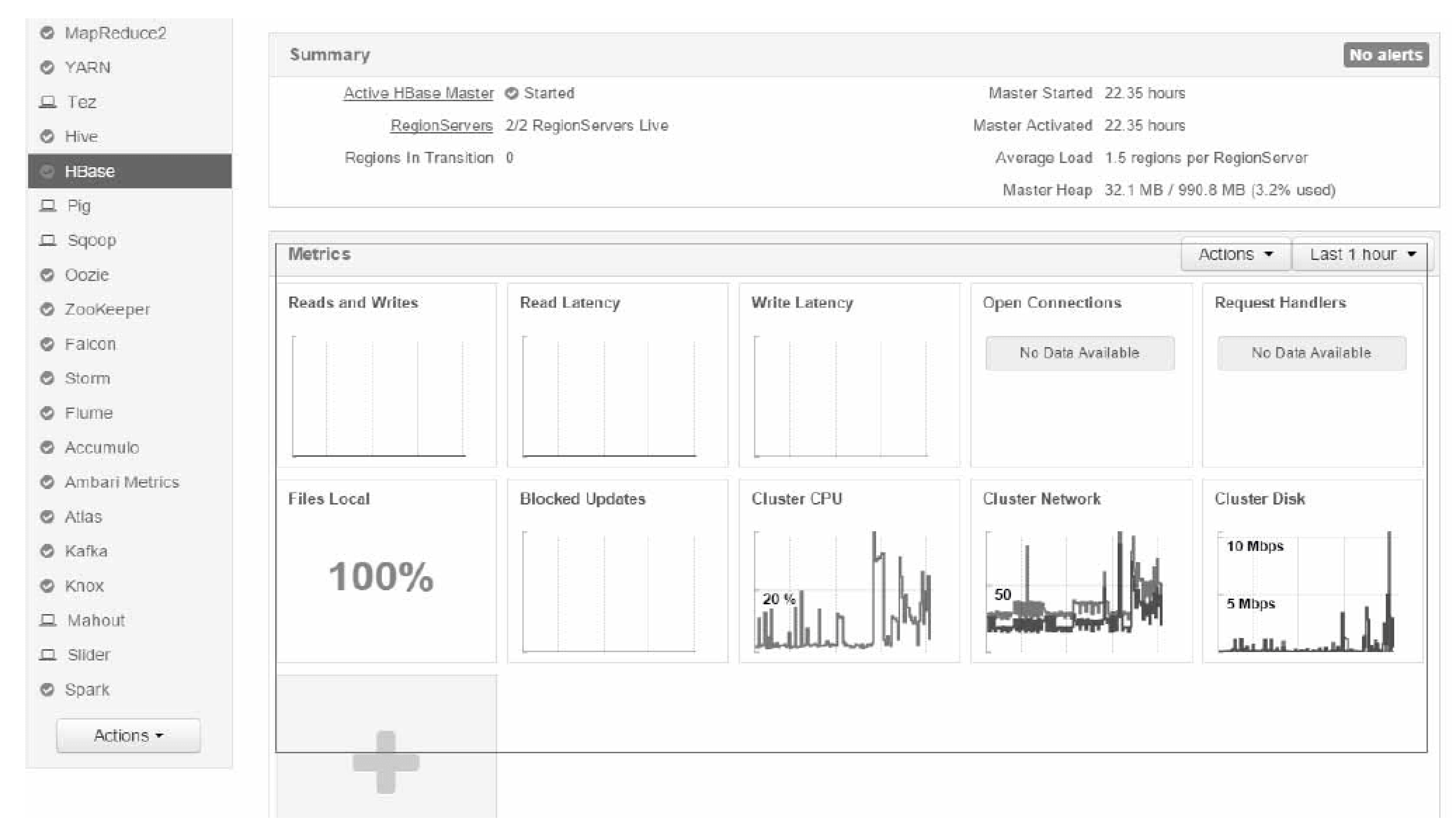


图 5-6　HBase 监控状态

可添加的小插件如图 5-7 所示。

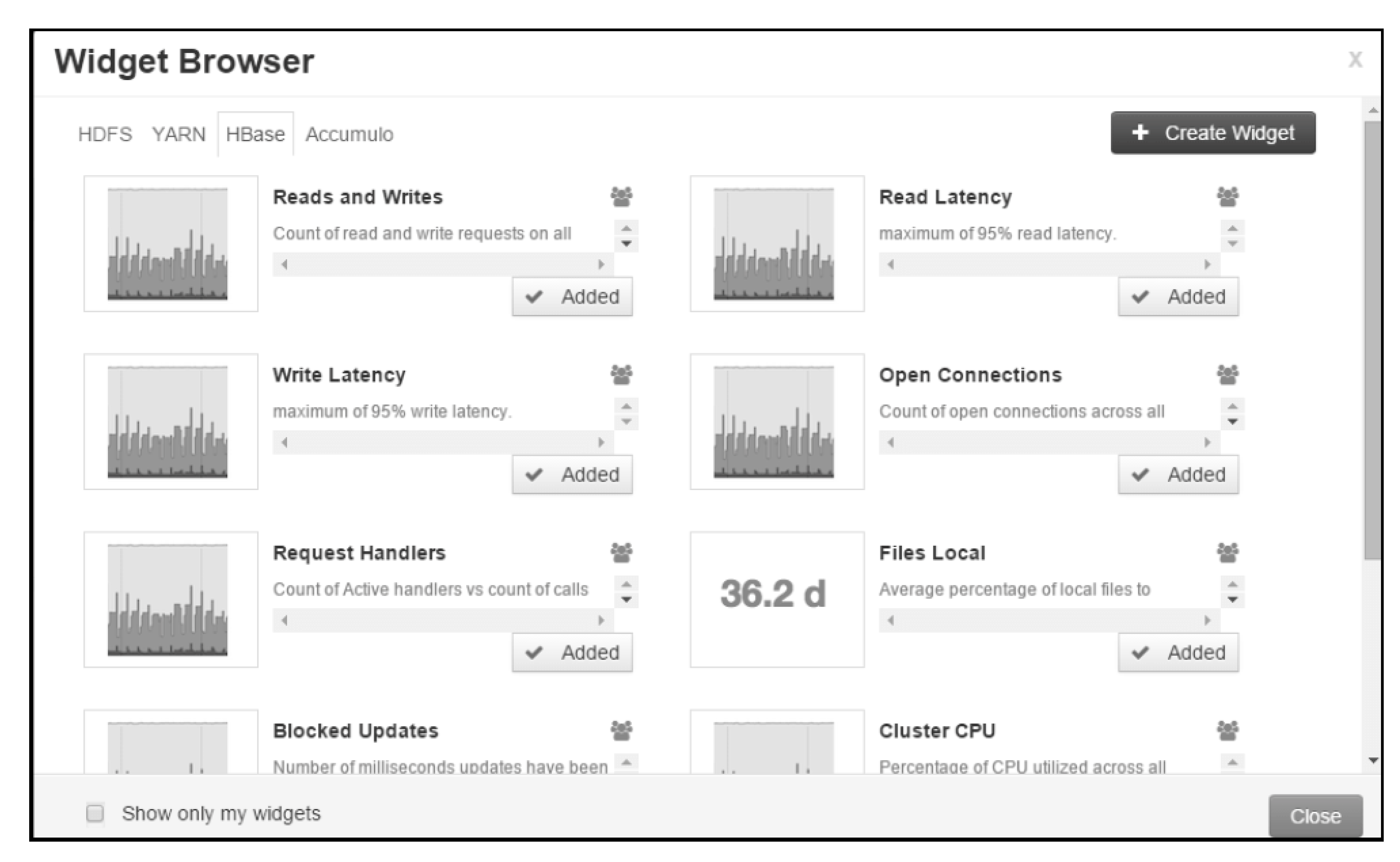


图 5-7　可添加插件

步骤 04　输入“hbase shell”进入 HBase 数据库，如图 5-8 所示。

```
[root@master ~]# hbase shell
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/2.3.0.0-2557/hadoop/lib/slf4j-log4j12
-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/2.3.0.0-2557/zookeeper/lib/slf4j-log4
j12-1.6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.1.1.2.3.0.0-2557, r6a55f21850cfccf19fa651b9e2c74c7f99bbd4f9, Tue Jul 1
4 09:41:13 EDT 2015

hbase(main):001:0>
```

图 5-8　HBase shell 界面

步骤 05　登录 HBase 数据库，验证 HBase 是否正常。输入 list 命令，列出 HBase 中的所有表，如图 5-9 所示。

```
hbase(main):001:0> list
TABLE
ambarismoketest
1 row(s) in 0.8650 seconds

=> ["ambarismoketest"]
```

图 5-9　Hbase shell 示例

如输出 HBase 中的表，则 HBase 正常。

步骤 06　退出 HBase Shell，输入 exit 即可。

5.2.4　HBase Shell 工具

下面，我们通过 HBase 命令行工具来做一些常用的 HBase 操作。

1. status 命令（查看 HBase 状态）

```
hbase(main):001:0> status
2 servers, 0 dead, 161.0000 average load
```

从上面的返回信息看出，该集群有 2 个 RegionServer。平均每台 RegionServer 有 161 个 Region。

2. version 命令

```
hbase(main):002:0> version
1.1.1.2.3.0.0-2557,  r6a55f21850cfccf19fa651b9e2c74c7f99bbd4f9,  Tue
Jul 14 09:41:13 EDT 2015
```

从上面的返回结果看出，一共包含三个部分，用逗号分隔。第一部分是版本号，第二部分是版本修订号，第三部分是编译 HBase 的时间。

3. create 命令

创建一个名为 test 的表，这个表只有一个列为 cf。其中表名和列簇都要用单引号括起来，并以逗号隔开。

```
hbase(main):003:0> create 'test','cf'
0 row(s) in 13.3570 seconds
=> Hbase::Table - test
```

创建语句可以有不同的格式，比如，下面的语句创建了 yang 表，列簇为 f1，该列簇的版本数为 5。

```
create  'yang',  {NAME=>'f1', VERSIONS=>5}
```

在上面的格式中，“=>”表示赋值，字符串使用单引号引起来。如果指定的列簇有特定的属性，需要使用花括号括起来。

4. list 命令

查看当前 HBase 中具有哪些表。

```
hbase(main):004:0> list
TABLE
test
yang
2 row(s) in 0.0820 seconds
```

5. put 命令

使用 put 命令向表中插入数据，参数分别为表名、行名、列名和值，其中列名前需要列簇为前缀，时间戳由系统自动生成。格式为：“put 表名,行名,列名([列族:列名]),值”。在下面的例子中，我们加入三行数据，第一行的行键为“row1”，列为 cf:a（列簇 cf 和列名 a），值为 value1。总共插入 3 条记录：

```
hbase(main):005:0> put 'test','row1','cf:a','value1'
0 row(s) in 0.8880 seconds
hbase(main):006:0> put 'test','row2','cf:b','value2'
0 row(s) in 0.0690 seconds
hbase(main):007:0> put 'test','row3','cf:c','value3'
0 row(s) in 0.0720 seconds
```

6. describe 命令

简单就一个 describe 命令用法，比如：查看表“test”的结构：

```
hbase(main):008:0> describe 'test'
Table test is ENABLED
test
COLUMN FAMILIES DESCRIPTION
{NAME => 'cf', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION_
SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL =
> 'FOREVER', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536', IN_MEMORY => '
false', BLOCKCACHE => 'true'}
1 row(s) in 0.3110 seconds
```

7. 查询数据（get 命令）

查看表“test”中的行键“row1”的相关数据：

```
get 'test','row1'
```

获取表为 test 里面的行键为 row1，列为 cf 的所有数据：

```
get 'test','row1','cf'
get 'test','row1',{COLUMN=>'cf'}
```

获取表为 test 里面的行键为 row1，列簇为 cf 里的字段为 a 的所有数据：

```
get 'test','row1','cf:a'
```

你还可以通过 timestamp 来获取版本的数据：

```
get 'test','row3',{COLUMNS=>'cf:c',TIMESTAMP=>1388641400138}
```

8. 全表查询

```
scan 'test'
```

Scan 命令可以带上过滤条件，比如：

```
scan 'test' , {COLUMNS=>'C1'}
scan 'test', {COLUMNS=>['C1:a','C2:b'],LIMIT=>2}
```

后一个命令指定了多列，限定了返回行数。关于更多的过滤条件，请参见 5.3.1 小节中的过滤条件内容。

9. 更新一条记录

```
put 'test','row2','cf:b' ,'newValue2'
put 'test','row3','cf:c' ,'newValue3'
```

10. 删除数据 delete 命令

删除行键为 row1 的行的“cf:a”字段：

```
delete 'test,'row1','cf:a'
```

删除整行：

```
deleteall 'test','row1'
```

11. 查询表中有多少行

```
count 'test'
```

12. 将整张表清空

```
truncate 'test'
```

HBase 是先将表 disable，然后 drop 表后重建表来实现 truncate 的功能。

13. 删除表

```
disable 'test'
drop 'test'
```

14. 自增

```
create 'table1', 'cf1', 'cf2'
incr 'table1', 'row1', 'cf1:count',1
incr 'table1', 'row1', 'cf1:count',1
incr 'table1', 'row1', 'cf1:count',10 //加10
incr 'table1', 'row1', 'cf1:count'  //不写的情况下就是等于加1
get_counter 'table1', 'row1', 'cf1:count'
```

在上面的代码中，create 创建了 table1 表，incr 是计算器操作的命令，对应的字段是 cf1:count，get_counter 返回计算器的值（13）。在 HBase 表中，计算器是以一个列（字段）的形式存在的。如果你用 scan 命令扫描这个表，你就会发现这个列。

表 5-3 总结了 HBase 的常用命令集。

表 5-3　HBase 的常用命令集

常用命令	描写
create	创建表
truncate	清空表，相当于重新创建指定表
describe	显示表相关的详细信息
alter	修改列簇模式
put	向指定的表单元中添加值
incr	增加指定表、行或列的值
get	获取行或单元（cell）的值
delete	删除指定的对象值（可以为表、行、列对应的值）
count	统计表中的行数
exists	测试表是否存在
list	列出 HBase 中存在的所有表
scan	通过对表的扫描来获取对应的值
disable	使表无效
drop	删除表
enable	使表有效

（续表）

常用命令	描写
status	返回 HBase 集群的状态信息
shutdown	关闭 HBase 集群
exit	退出 HBase shell
tools	列出 HBase 所支持的工具
version	返回 HBase 版本信息
whoami	查看用户身份
hbck	文件检测修复工具
hfile	文件查看工具
hlog	日志查看工具
export	数据导出工具
import	数据导入工具

除了使用 Shell 工具和下节的 Java API 来操纵 HBase 之外，HBase 提供了 Web UI 来查看 HBase 的实时状态信息。

5.3 HBase 编程

HBase 提供了对大规模数据的随机、实时读写访问。HBase 是一个非关系型数据库，即 NoSQL 数据库，它不使用 SQL 作为查询语言，也避免使用 SQL 的 JOIN 操作。RDBMS 要求每个表都有固定的表模式（即：这个表有多少列，各个列的名称和数据类型都是固定的），而 HBase 的表模式可以不固定（即每一行的数据可以有不同的列）。HBase 无须事先为要存储的数据建立字段，允许随时添加字段。值是由行关键字、列关键字和时间戳确定。整个数据模型是 Schema → Table → Column Family → Column → RowKey → TimeStamp → Value。HBase 提供了丰富的 Java API 来操纵数据库上的数据。

5.3.1 增删改查 API

与 HBase shell 工具相对应，我们可以通过 Table 接口对表进行 Get、Put、Scan、Delete 等操作，从而完成向 HBase 存储和检索数据，删除数据等操作。比如：Table 接口中提供了 get()方法，返回一行或多行数据。如图 5-10 所示，HBase 提供了丰富的 API 来操纵 HBase 数据库。

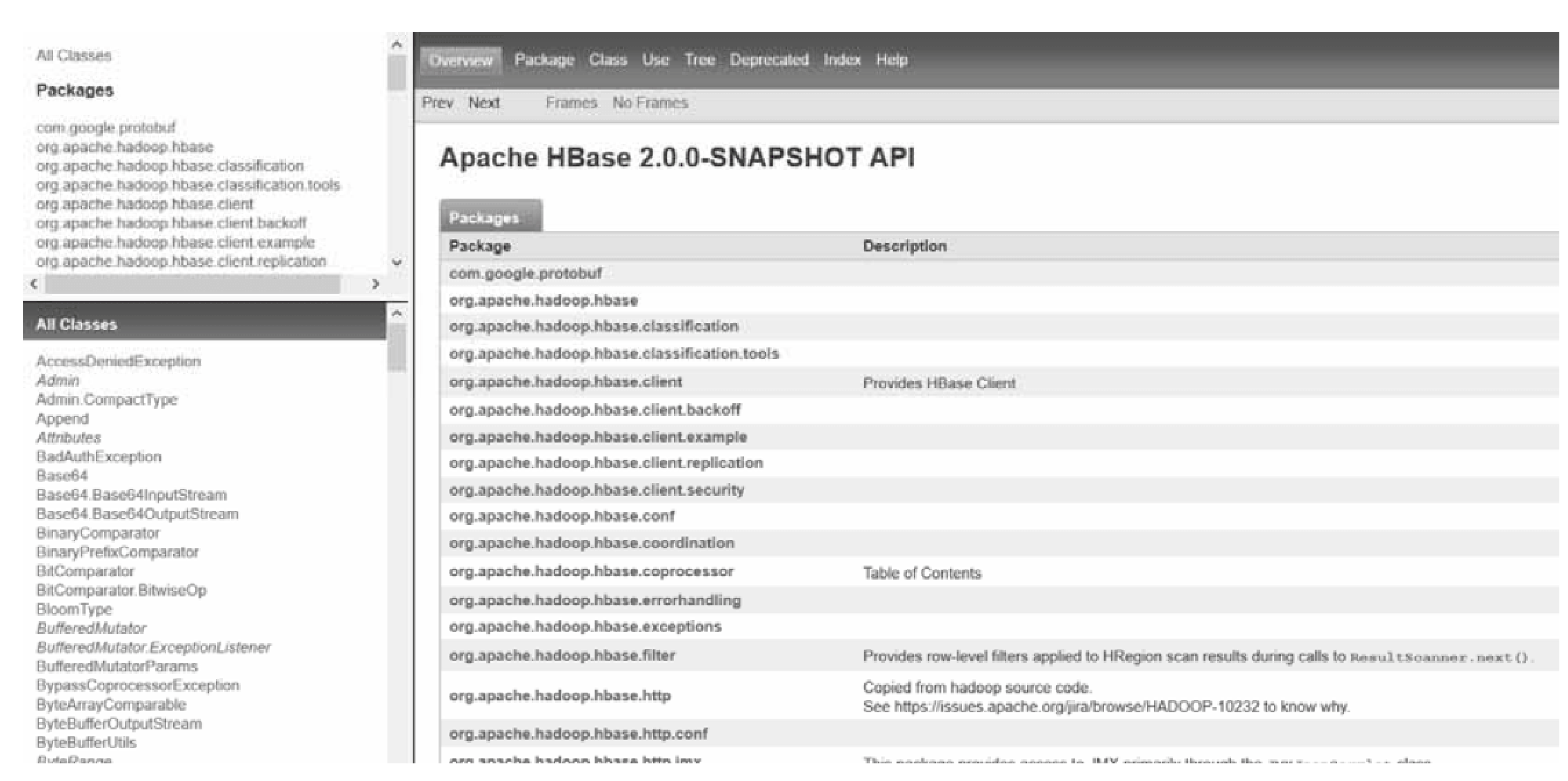


图 5-10　HBase JAVA API

下面我们说明常用的 HBase API 所提供的类的功能，以及它们之间有什么样的关系。

1. org.apache.hadoop.hbase.HBaseConfiguration

作用：通过此类可以对 HBase 进行配置。

用法实例： Configuration config = HBaseConfiguration.create();

说明： HBaseConfiguration.create() 默认会从 classpath 中查找 hbase-site.xml 中的配置信息，初始化 Configuration。

2. org.apache.hadoop.hbase.client.Connection

作用：Connection 是一个接口，它的对象代表着到 HBase 的一个数据库连接。我们使用 ConnectionFactory.createConnection(config)创建一个 Connection。通过这个 Connection 实例，可以使用 Connection.getTable()方法取得 Table 对象，例如：

```
Connection connection = ConnectionFactory.createConnection(config);
Table table = connection.getTable(TableName.valueOf("table1"));
try {
   // 使用 table 对象
} finally {
   table.close();
   connection.close();
}
```

3. org.apache.hadoop.hbase.client.Table

作用：这个接口可以和 HBase 进行通信，对表进行操作。

用法：Table tab = connection.getTable(TableName.valueOf("table1"));

ResultScanner sc = tab.getScanner(Bytes.toBytes(“familyName”));

说明：获取表内列簇 familyNme 的所有数据。HBase 是大数据的分布式数据库，当使用全表扫描肯定是不合理的。我们可以给 Scan 操作指定 startRow 参数来定义扫描读取 HBase 表的起始行键，同时也可选用 stopRow 参数来限定读取到何处停止。Scan 操作的结果被封装在一个 ResultScanner 对象中：

```
    Scan s = new Scan();
    //通过 rowkey 来指定数据开始和结束
    s.setStartRow(Bytes.toBytes("2016-3-22"));
    s.setStopRow(Bytes.toBytes("2016-3-23"));
    rs = table.getScanner(s);
```

这个 Table 接口包含了表的增删改查的 API:

```
    //删除
    boolean checkAndDelete(row,family,qualifier,value,Delete);
    boolean checkAndDelete(row,family,qualifier,
CompareFilter.CompareOp ,value,Delete);
    void delete(Delete);
    void delete(List<Delete>);//支持批量删除
    //查询
    Result get(Get);
    Result[] get(List<Get>);//支持批量查询
    ResultScanner getScanner(family,qualifier);
    ResultScanner getScanner(Scan);
    //插入
    boolean checkAndPut(row,family,qualifier,value,Put);
    boolean checkAndPut(row,family,qualifier,
CompareFilter.CompareOp ,value,Put);
    void put(List<Put>);//支持批量插入
    void put(Put);
    //验证是否存在
    boolean exists(Get);
```

4. org.apache.hadoop.hbase.client.Put

作用：添加一行数据。

用法：Put p = new Put(row);

p.add(family,qualifier,value);

table.put(p);

说明：构造一个 Put 对象（参数为行键），该对象封装了要插入的数据。添加 “family（列

簇）,qualifier（列名）,value（值）”指定的值。Table 接口的 put 方法要么向表插入新行（如果行键是新的），要么更新行（如果行键已经存在）。可以一次向表中插入一行数据，也可以一次操作一个集合，同时向表中写入多行数据。要注意的是，HBase 没有 Update 操作，这是通过 Put 操作完成对数据的修改的。Put 操作会为一个 Cell 创建一个版本，默认为当前的时间戳，我们也可以自己设置时间戳。

5. org.apache.hadoop.hbase.client.Get

作用：获取单个行的数据。

用法：Get get = new Get(row);

get.addColumn(family,qualifier);//可以不指定列信息，则返回所有列

Result result = table.get(get);

说明：构造 Get 对象，该对象封装了要查询的行键、列簇和列名。执行查询后就获取了表中 row 行的对应数据。默认情况下，get()方法一次取回该行全部列的数据，我们可以限定只返回某个列簇对应的列的数据，或者进一步限定为某些列的数据。正如上面例子中所看到的，该方法返回的数据将被封装在一个 Result 对象中。用 Result 类提供的方法，可以从服务器端获取匹配指定行的特定返回值，这些值包括列簇、列限定符和时间戳等。我们可以在 Get 对象上设置版本信息，用于返回老版本。比如，下面的代码设置了要返回最近 3 个版本：

```
get.setMaxVersions(3);
```

由于 HBase 按列存储特性，所以，按照行键的查询的速度非常快，应该在毫秒级别。按照行键查询是 HBase 检索中最常用，并且是速度最快的查询。

6. org.apache.hadoop.hbase.client.ResultScanner

作用：获取多行数据（也叫扫描读）。

用法：
```
ResultScanner scanner = table.getScanner(family);
For(Result rowResult : scanner){
        Bytes[] str = rowResult.getValue(family,column);
}
```

说明：循环获取行中列值。

7. org.apache.hadoop.hbase.client.Scan

作用：获取多行数据（也叫扫描读）。

用法：
```
Scan scanner   = new Scan();
scanner.setTimeRange(startTime,endTime);//设置时间段
scanner.setStartRow(startRow);
scanner.setSTopRow(stopRow);
scanner.addColumn(family,column);
ResultScanner rsScanner = table.getScanner(scanner);
```

```
        For(Result rowResult : scanner){
                Bytes[] str = rowResult.getValue(family,column);
}
```

说明：循环获取行中列值。上面的扫描器 scan.setStartRow(Bytes)和 scan.setStopRow(Bytes)是用来设置要查询的数据的行键的范围。

8. org.apache.hadoop.hbase.client.Delete

作用：这是 HBase 的 JAVA API 中删除数据的类。我们可以通过多种方法限定要删除的数据。与 RDBMS 不同，HBase 可以删除某一个列簇、某个列、某个单元，或者指定某个时间戳，删除比这个时间早的数据。比如：

```
Delete del= new Delete(row1);
del.addColumn(family, qualifier);
table.delete(del);
```

在上面的例子中，构造 Delete 对象，封装所要删除的行键和列信息，然后执行删除操作。

5.3.2 过滤器

上节中的操作过于简单，有时就不能满足复杂查询的需求。这时候就需要更加高级的过滤器（Filter）来查询了。前面的 Get 和 Scan 类都可以配置过滤器，方法为 setFilter(filter)。过滤器可以根据列簇、列、版本等更多的条件来对数据进行过滤。基于 HBase 本身提供的三维（行键、列、版本），这些过滤器可以更高效地完成过滤功能。过滤器是在 RegionServer 上发挥作用，所以，过滤器可以减少网络传输的数据。下面我们来看一下具体的过滤器类。

1. org.apache.hadoop.hbase.filter.FilterList

作用：FilterList 代表一个过滤器列表，过滤器之间具有 FilterList.Operator.MUST_PASS_ALL（就是 AND）和 FilterList.Operator.MUST_PASS_ONE（就是 OR）的关系，下面展示一个过滤器的“或”关系，检查同一属性的“value1”或“value2”。

```
FilterList list = new FilterList(FilterList.Operator.MUST_PASS_ONE);
SingleColumnValueFilter filter1 = new
SingleColumnValueFilter(Bytes.toBytes("cfamily"),
Bytes.toBytes("column"),CompareOp.EQUAL,Bytes.toBytes("value1"));
list.add(filter1);
SingleColumnValueFilter filter2 = new
SingleColumnValueFilter(Bytes.toBytes("cfamily"), Bytes.toBytes(“column”),
CompareOp.EQUAL, Bytes.toBytes("value2"));
list.add(filter2);
```

2. org.apache.hadoop.hbase.filter.SingleColumnValueFilter

作用：SingleColumnValueFilter 类是一个列值过滤器，用于测试列值是否相等（CompareOp.EQUAL），不等（CompareOp.NOT_EQUAL），或范围（比如：CompareOp.GREATER）。下面的例子检查了列值和字符串“values”是否相等：

```
    SingleColumnValueFilter filter = new
SingleColumnValueFilter(Bytes.toBytes("cFamily"), Bytes.toBytes("column"),
CompareOp.EQUAL, Bytes.toBytes("values"));
    scan.setFilter(filter);
```

3. org.apache.hadoop.hbase.filter.ColumnPrefixFilter

作用：ColumnPrefixFilter 用于返回只与指定列名的前缀相等的那些行。在 HBase 中，每行的列的个数可能是不同的。比如：下面是查找以“yang”为前缀的所有列的值：

```
    byte[] prefix = Bytes.toBytes(“yang”);
    Filter f = new ColumnPrefixFilter(prefix);
    Scan scan = new Scan(row,row); //限制为一行
    scan.addFamily(family);//限制为一个列簇
    scan.setFilter(f);
```

4. org.apache.hadoop.hbase.filter.MultipleColumnPrefixFilter

作用：MultipleColumnPrefixFilter 和 ColumnPrefixFilter 行为差不多，但可以指定多个前缀。比如：下面指定了“yang”和“zhenghong”两个前缀：

```
    byte[][] prefixes = new byte[][] {Bytes.toBytes("yang"),
Bytes.toBytes("zhenghong")};
    Filter f = new MultipleColumnPrefixFilter(prefixes);
    Scan scan = new Scan(row, row);
    scan.addFamily(family);
    scan.setFilter(f);
```

5. org.apache.hadoop.hbase.filter.ColumnRangeFilter

作用：ColumnRangeFilter 过滤器可以进行列名内部扫描。比如：下面的例子扫描所有在“a-100”和“b-999”之间的列：

```
    byte[] startColumn = Bytes.toBytes(“a-100”);
    byte[] endColumn = Bytes.toBytes(“b-999”);
    Scan scan = new Scan(row,row);
    scan.addFamily(family);
    Filter f = new ColumnRangeFilter(startColumn, true, endColumn,true);
    scan.setFilter(f);
```

6. org.apache.hadoop.hbase.filter.QualifierFilter

作用：QualifierFilter 是基于列名的过滤器。比如：

```
    Filter f = new QualifierFilter(CompareFilter.CompareOp.EQUAL,
"QualifierName");
    scan.setFilter(f);
```

7. org.apache.hadoop.hbase.filter.RowFilter

作用：RowFilter 是行键过滤器。一般而言，执行 Scan 时使用 startRow/stopRow 方式比较好。这个行键过滤器完成对某一行的过滤。比如：

```
    Filter f = new RowFilter(CompareFilter.CompareOp.EQUAL,
    new BinaryComparator(Bytes.toBytes("row-A")));
    scan.setFilter(f);
```

8. org.apache.hadoop.hbase.filter.PageFilter

作用：PageFilter 用于按行分页。比如：

```
    Filter filter = new PageFilter(15);
    Scan scan = new Scan();
    scan.setFilter(filter);
```

比较器是过滤器的核心组件之一，用于处理具体的比较逻辑,下面就是各个比较器类：

1. org.apache.hadoop.hbase.filter.RegexStringComparator

作用：RegexStringComparator 是支持正则表达式的比较器。

过滤器与比较器一起使用会很方便。下面的代码中的参数 reg 就是正则验证的规则。

```
    Scan scan = new Scan();
    String reg = "^188([0-9]{8})$";//满足188开头的手机号
    RowFilter filter = new RowFilter(CompareOp.EQUAL, new
RegexStringComparator(reg));
    scan.setFilter(filter);
    ResultScanner rs = table.getScanner(scan);
    for(Result rr : rs){
    for(KeyValue kv : rr.raw()){
            ...
    }
    }
```

2. org.apache.hadoop.hbase.filter.SubstringComparator

作用：SubstringComparator 用于检测一个子串是否存在于列（单元）值中。大小写不敏

感。比如：

```
    //检测“zhenghong”是否存在于查询的列值中
    SubstringComparator comp = new SubstringComparator("zhenghong");
    SingleColumnValueFilter filter = new SingleColumnValueFilter(cf,
column,CompareOp.EQUAL, comp);
    scan.setFilter(filter);
```

3. org.apache.hadoop.hbase.filter.BinaryPrefixComparator

作用：BinaryPrefixComparator 是前缀二进制比较器，它只比较前缀是否相同。

4. org.apache.hadoop.hbase.filter.BinaryComparator

作用：BinaryComparator 是二进制比较器，用于按照字典顺序比较 Byte 数据值，比如：

```
    Filter  filter1  =  new  ValueFilter(CompareFilter.CompareOp.NOT_EQUAL,
new BinaryComparator(Bytes.toBytes(“val=0” )));
    Scan scan = new Scan();
    scan.setFilter(filter1);
    ResultScanner scanner1 = table.getScanner(scan);
```

5.3.3　计数器

Hbase 提供一个计数器工具，可以方便快速地进行计数的操作，从而免去了加锁等保证原子性的操作。实质上，计数器还是列，有自己的簇和列名。值得注意的是，维护计数器值的最好方法是使用 HBase 提供的 API，直接操作更新很容易引起数据的混乱。计数器的增量可以是正数或负数，正数代表加，负数代表减。计数器在 RegionServer 上完成。

org.apache.hadoop.hbase.client.Table 接口提供了 incrementColumnValue(byte[] row, byte[] family, byte[] qualifier, long amount)方法在某一行某一列上增加值。比如：

```
    long cur = table.incrementColumnValue(rowkey1,cf1,c1,0L);//不增加
    cur = table.incrementColumnValue(rowkey1,cf1,c1,1L);//增加1
    cur = table.incrementColumnValue(rowkey1,cf1,c1,10L);//增加10
```

多列计数器需要使用计数器的类，即：org.apache.hadoop.hbase.client.Increment。首先使用 Increment 的构造方法构造一个 Increment 实例，然后使用这个类的 addColumn (byte[] family, byte[] qualifier, long amount)方法指定一个计数器列。多次调用这个方法可以添加多个列。比如：

```
    Increment incr = new Increment(Bytes.toBytes("rk1"));
    incr.addColumn(Bytes.toBytes("cf1"),Bytes.toBytes("count1"),1);
    incr.addColumn(Bytes.toBytes("cf1"),Bytes.toBytes("count2"),1);
    incr.addColumn(Bytes.toBytes("cf2"),Bytes.toBytes("count2"),5);
```

```
incr.addColumn(Bytes.toBytes("cf2"),Bytes.toBytes("count2"),5);
Result result1 = table.increment(incr);
```

5.3.4 原子操作

HBase 提供基于单行数据操作的原子性保证。即：对同一行的变更操作（包括针对一列/多列/多列簇的操作），要么完全成功，要么完全失败，不会有其他状态，比如：

```
A 客户端： 针对 rowkey=100的行发起操作：cf1:a = 1  cf2:b=1
B 客户端： 针对 rowkey=100的行发起操作：cf1:a = 2  cf2:b=2
```

cf1 和 cf2 为列簇，a 和 b 为列。A 客户端和 B 客户端同时发起请求，最终 rowkey=100 的行的各个列的值可能是 cf1:a = 1 cf2:b=1，也可能是 cf1:a = 2 cf2:b=2，但绝对不会是 cf1:a = 1 cf2:b=2。HBase 基于行锁来保证了单行操作的原子性。还有，Table 接口提供了 checkAndPut()和 checkAndDelete 方法，这两个方法在维持原子操作的同时还提供了更精细的控制，比如：tab.checkAndPut(rowKey,family,column,value,put)检查这个列值是否等于指定的值 value（即：值没有发生改变），如果等于则 put 新值。

5.3.5 管理 API

org.apache.hadoop.hbase.client.Admin 是一个管理的接口，可以管理 HBase 数据库中的表。它通过 Connection.getAdmin()获得 Admin 的一个实例。通过这个接口，我们可以创建表、删除表、修改表、查询表的信息等。还可以管理 Region，分割与合并等操作。

下面是涉及表的创建、删除、修改的方法。创建表的方法都以 create 开头，删除都以 delete 开头。

```
//创建
void createTable(HTableDescriptor desc);
//删除
void deleteTable(TableName tableName);
HTableDescriptor[] deleteTables(String regex);
HTableDescriptor[] deleteTables(Pattern pattern);
//修改
void addColumnFamily(TableName tableName, HColumnDescriptor
columnFamily);
void modifyColumnFamily(TableName tableName, HColumnDescriptor
columnFamily);
void modifyTable(TableName tableName, HTableDescriptor htd);
//修改表的状态，获取表的信息
```

```
void disableTable(TableName tableName);
void enableTable(TableName tableName);
HTableDescriptor getTableDescriptor(TableName tableName);
boolean isTableDisabled(TableName tableName);
Boolean isTableEnabled(TableName tableName);
TableName[] listTableNames();
```

在上面的 API 中，有以下两个描述表和列的类：org.apache.hadoop.hbase.HTableDescriptor 包含了表的名字以及表的列簇信息，org.apache.hadoop.hbase.HColumnDescriptor 维护列簇的信息。用法如下：

```
HTableDescriptor htd =new HTableDescriptor(tablename);
htd.addFamily(new HColumnDescriptor("myFamily"));
admin.createTable(htd);
```

使用 Java API 删除一个表时，需要包含 2 个步骤，第一步是 disableTable(tableName)，第二步是 deleteTable（tableName）。第一个 API 禁用表，第二个 API 删除表。下面的例子演示了如何添加、删除和修改列信息：

```
if (modCols.size()>0 || addCols.size()>0 || delCols.size()>0 ) {
 for (final HColumnDescriptor col : modCols ) {
     admin.modifyColumnFamily(tableName, col);
 }
 for (final HColumnDescriptor col : addCols ) {
     admin.addColumnFamily(tableName,col);
 }
 for (final HColumnDescriptor col : delCols ) {
     admin.deleteColumnFamily(tableName,col);
   }
}
```

HBase 也提供了类似触发器和存储过程的功能，这是通过协处理器完成的，协处理器提供的 Observer 类似 RDBMS 的触发器，而协处理器提供的 EndPoint 类似存储过程。关于这方面的具体内容，可参考 HBase 文档。

5.4 其他 NoSQL 数据库

Cassandra 和 Impala 是另外两个知名的 NoSQL 数据库。Cassandra 是一套开源分布式 NoSQL

数据库系统，能够在一堆普通的服务器上存储和管理数据。它最初由 Facebook 开发，用于存储和查询 Facebook 收件箱。Cassandra 既可用作实时的数据存储（比如：在线事务系统），也可用作大型的 BI 系统（这些系统往往是有大量的读操作）。Cassandra 的每个节点的角色是一样的，没有主从节点之分，所有节点彼此同等地通信。这个优势保证了 Cassandra 没有单点失败的问题。由于 Cassandra 良好的可扩展性，它被苹果、Comcast、Instagram、Spotify、eBay、Netflix、Twitter 等知名公司所使用，成为了一种流行的分布式数据存储方案。使用 Cassandra 的最大的一个生产系统是在 75000 节点的集群之上操作 PB 级别数据。

Impala 是 Cloudera 公司主导开发的新型查询引擎，它提供 SQL 语义，能查询存储在 Hadoop 的 HDFS 和 HBase 中的 PB 级大数据。已有的 Hive 系统虽然也提供了 SQL 语义，但由于 Hive 底层执行使用的是 MapReduce 引擎，仍然是一个批处理过程，难以满足查询的交互性。相比之下，Impala 的最大优势就是它比 Hive 较少的延迟（SQL 的执行速度快）。读者需要注意的是，Hive 支持所有来自 Impala 的调用，但是，反之则不成立。

第 6 章 大数据访问：SQL引擎层

在上一章，我们提到了 HBase 的很多优点，比如，HBase 提供了海量数据的毫秒级查询。可见，HBase 是个非常好的实时查询框架，缺点就是查询功能非常薄弱，仅限于通过行键查询。还有一点是，许多程序员和 DBA 对 HBase 还是很不习惯，因为他们需要抛弃从前关系数据库的很多常识来学习如何使用 HBase。那么，有没有一种方法操作 HBase，就像操作传统的关系型数据库一样呢？这就是本章我们要阐述的 Phoenix，它提供了 HBase 的 SQL 访问功能，可以使用标准的 JDBC API 操作去创建表、插入记录、查询数据。Phoenix 是一个 Java 中间层，可以让开发者在 Apache HBase 上执行 SQL 查询。将 SQL 查询转换为一个或多个 HBase scan，并编排执行以生成标准的 JDBC 结果集。如果我们手工使用 HBase 的 API 去写这些代码，也会得到相同的运行结果和执行速度。但是，使用 phoenix 的效果却会带来更快的开发效率。与直接使用 HBase API、协同处理器与自定义过滤器相比，对于简单查询来说，Phoenix 性能量级是毫秒，对于百万级别的行数来说，其性能量级是秒。

大数据时代的信息爆炸，使得分布式/并行处理变得如此重要。无论是传统行业，还是新兴行业（比如：互联网行业），海量数据需要更大的硬件资源来处理。从单机应用到集群应用的发展中，诞生了 Hadoop MapReduce 这样的分布式框架，简化了并行程序的开发，提供了水平扩展和容错能力。虽然 MapReduce 的应用非常广泛，但这类框架的编程接口仍然比较低级，编写复杂处理程序或 Ad-hoc 查询仍然十分耗时，并且代码很难复用，学习成本太高。目前，Google、Facebook 等公司都在底层分布式计算框架之上又提供更高层次的编程模型，将开发者不关心的细节封装起来，提供了更简洁的编程接口。Hive 就是这样的一种工具，它提供了类似 SQL 的 Hive 查询语言（简称 HiveQL）来查询 HDFS 和 HBase 上的数据。Hive 可以将这些查询转换为 MapReduce 任务，从而让开发人员使用熟悉的 SQL 来开发 Hadoop 应用系统。它最先是由 Facebook 开发并贡献出来的，现在包括 Netflix 等公司都在使用和更新这个系统。

本章主要讲解 Phoenix 和 Hive。我们统称为大数据访问的 SQL 引擎层。

6.1 Phoenix

Phoenix 是由 Salesforce 公司开源提供给 Apache。Phoenix 查询引擎会将 JDBC API 编译成一

系列的 HBase 的 scan 操作和服务器端的过滤器，执行后生成标准的 JDBC 结果集返回。本质上讲，Phoenix 就是能够让开发人员使用 SQL 和 JDBC 来访问 HBase。因此，Phoenix 是构建在 Apache HBase（列式大数据存储）之上的一个 SQL 中间层，它完全使用 Java 编写，类似一个内嵌在客户端的 JDBC 驱动程序。Phoenix 对于程序员和 DBA 来说，是一个不用学习 HBase 就可以进行开发和管理 HBase 的好工具。有了 Phoenix，你就可以使用最熟悉的 SQL 语句和 JDBC API 对数据进行查询、增加、修改和删除了。Phoenix 的官网是 http://phoenix.apache.org。使用 Phoenix 开发 JDBC 程序同一般的 JDBC 程序没有太大区别。下面是 Phoenix 的一些特性：

- 嵌入式的 JDBC 驱动，实现了大部分的 java.sql 接口，包括元数据 API;
- 可以通过多行键或是键/值单元对列进行建模;
- 完善的查询支持，可以使用多个谓词以及优化的 scan;
- DDL 支持：通过 CREATE TABLE、DROP TABLE 及 ALTER TABLE 来创建表，并给表添加/删除列;
- DML 支持：用于逐行插入的 UPSERT VALUES、用于相同或不同表之间大量数据传输的 UPSERT SELECT、用于删除行的 DELETE;
- 从 4.7 版本开始支持事务，并紧跟 ANSI SQL 标准;
- Phoenix 将 Query Plan 直接使用 HBase API 实现，减少了查询的时间延迟。Phoenix 的 SQL Query Plan 基本上都是通过构建一系列 HBase Scan 来完成。

在 HBase 上提供 SQL 接口，有如下几个原因：

- 使用诸如 SQL 这样易于理解的语言可以使人们能够更加轻松地使用 HBase 。相对于学习另一套私有 API，人们可以使用熟悉的 SQL 语言来读写数据;
- 使用诸如 SQL 这样更高层次的语言来编写，减少了你所需编写的代码量。比如，使用 Phoenix 比使用原生的 HBase API 会少很多行代码;
- 加上 SQL 这样一层抽象层可以对查询进行大量优化。比如，对于 GROUP BY 查询来说，我们可以利用 HBase 中协同处理器这样的特性。借助于该特性，我们可以在 HBase 服务器上执行 Phoenix 代码。因此，聚合可以在服务端执行，而不必在客户端，这样会极大减少客户端与服务端之间传输的数据量。此外，Phoenix 还会在客户端并行执行 GROUP BY，这是根据行键的范围来截断扫描而实现的。通过并行执行，结果会更快地返回。所有这些优化都无须用户参与，用户只需发出查询即可。
- 通过使用业界标准的 API（如 JDBC），我们可以利用现有的工具来使用这些 API。比如，你可以使用现成的 SQL 客户端（如 Squirrel）连接 HBase 服务器并执行 SQL。

6.1.1 安装和配置 Phoenix

Phoenix 项目是构建在 Apache HBase 之上的一个 SQL 中间层，通过标准化的 SQL 语言来访问 HBase 数据，但是性能上不差。对于 10 万到 100 万的行的简单查询来说，Phoenix 要胜过 Hive。对于使用了 HBase API、协同处理器及自定义过滤器的 Impala 与 OpenTSDB 来说，进行

相似的查询，Phoenix 的速度也会更快一些。Phoenix 的官方下载地址为 http://phoenix.apache.org/download.html。在 HDP yum 源里有 Phoenix 安装包。安装步骤如下：

步骤 01　在 master 上运行：

```
yum install phoenix -y
```

步骤 02　在 slave 各节点，将 Phoenix-server.jar 包从 master 上复制到 hbase 的 lib 目录下。

步骤 03　重启 Hbase 完成安装。

为了在 Windows 上使用 Phoenix，你需要安装一个可视化控件 squirrel-sql。在 Windows 上安装 squirrel-sql 的步骤如下：

步骤 01　首先把这个安装包（squirrel-sql-3.5.2-standard.jar）拷贝到机器上。

步骤 02　进入命令行操作界面（cmd），如图 6-1 所示。

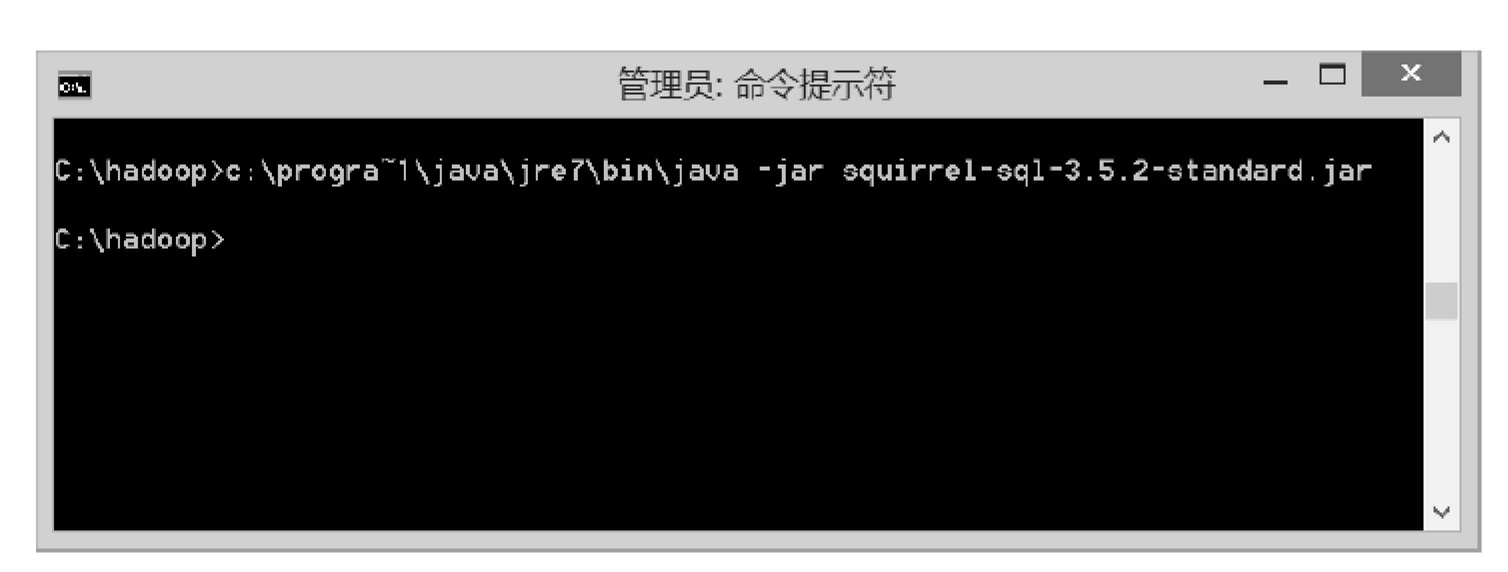


图 6-1　执行安装程序

步骤 03　在命令行窗口中，切换到包含安装包的位置，然后输入 java -jar squirrel-sql-3.5.2-standard.jar（如图 6-1 所示），安装程序开始执行。如图 6-2 所示，按照安装程序的提示，确定所安装的路径，然后一步步安装下去。

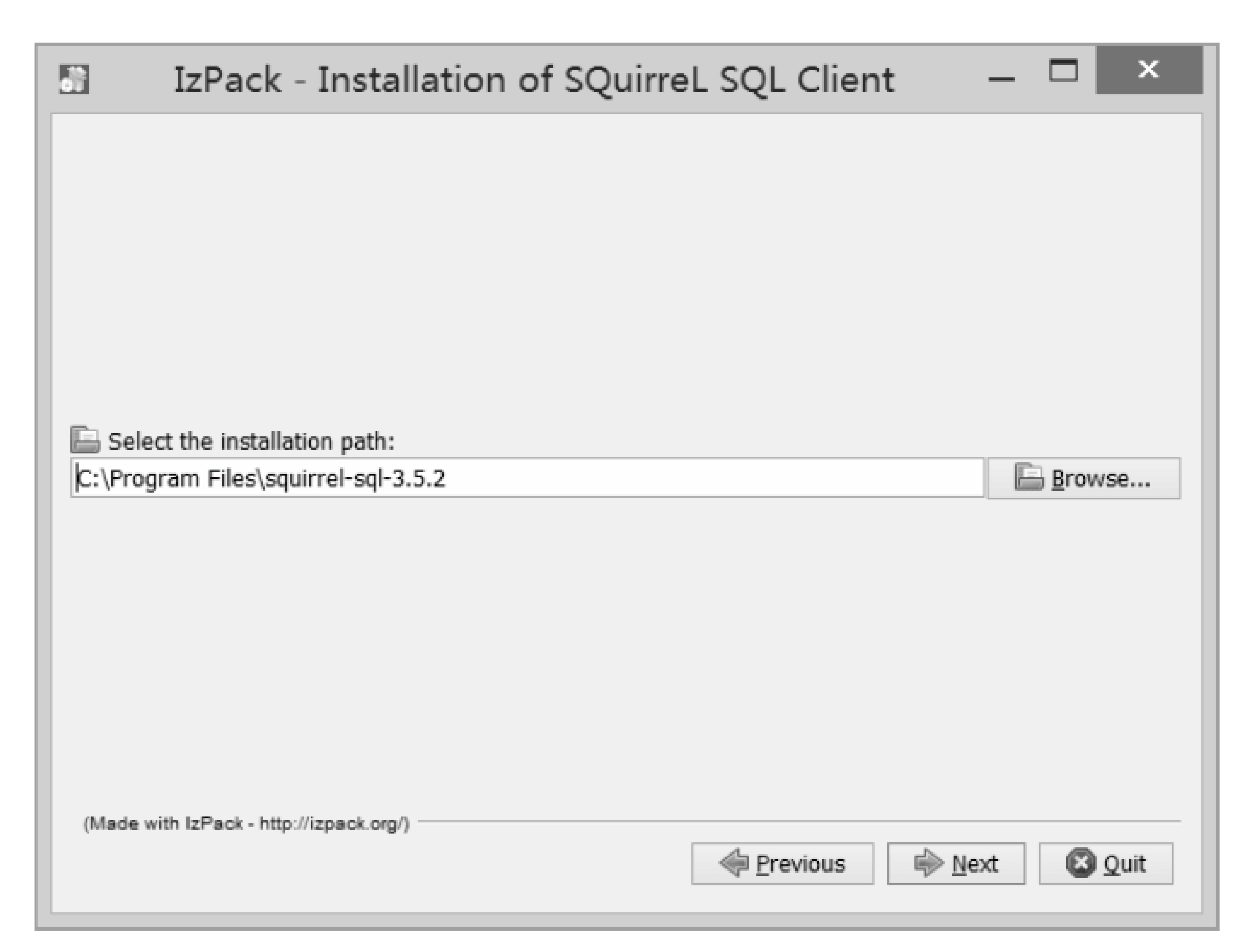


图 6-2　安装路径

步骤 04 拷贝 phoenix 的客户端 jar 包（phoenix-4.1.0-incubating-SNAPSHOT-client.jar 和 phoenix-core-4.1.0-incubating-SNAPSHOT）到安装目录的 lib 文件夹下（在我的机器上，是 C:\Program Files\squirrel-sql-3.5.2\lib）。

步骤 05 启动 squirrel，如图 6-3 所示。



图 6-3　SQuirreL 安装图标

步骤 06 下面开始配置 SQuirreL。在如图 6-4 的窗口上单击左侧的 Driver，单击加号，添加如图 6-5 所示的数据库驱动器。

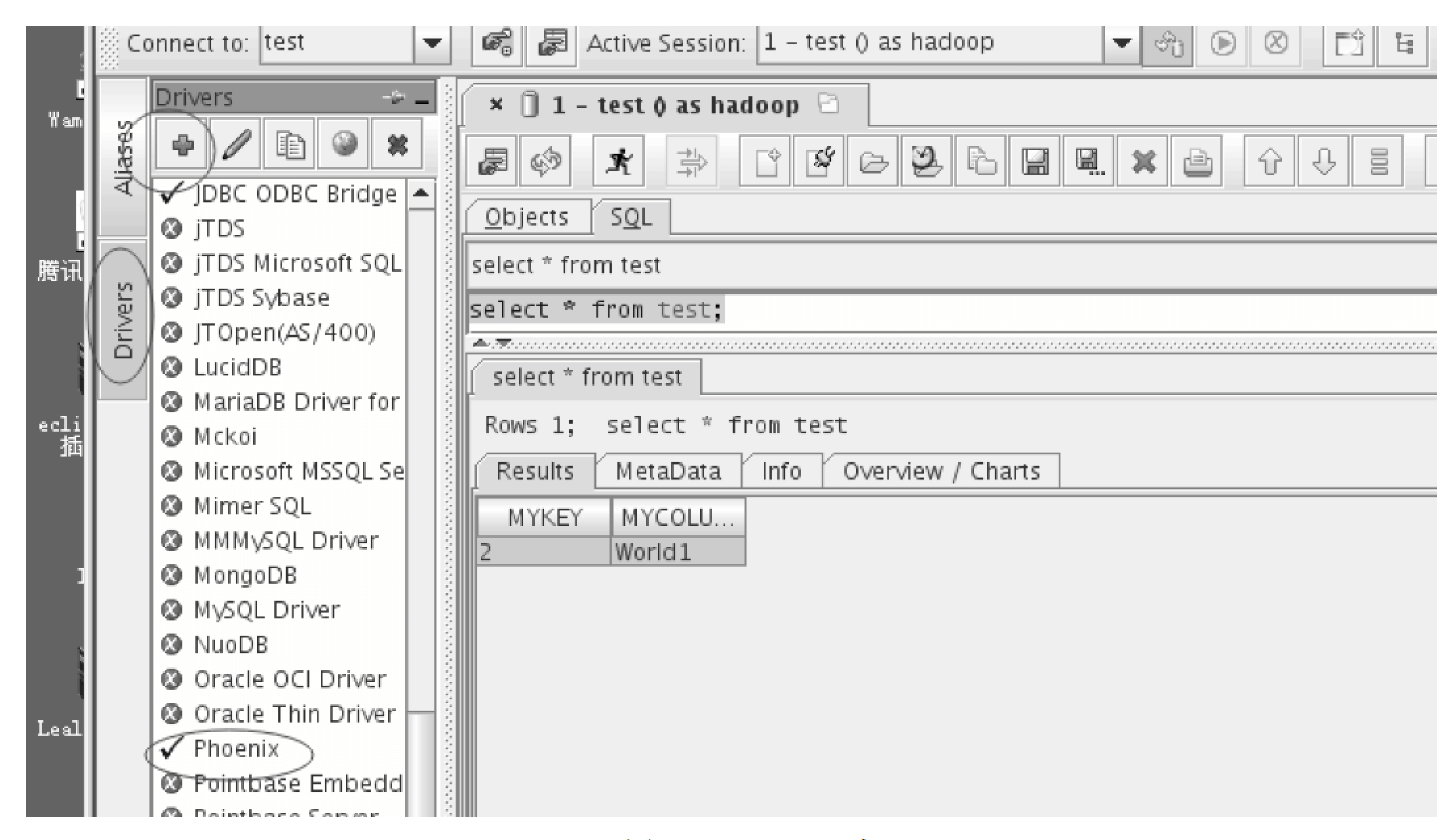


图 6-4　SQuirreL 窗口

步骤 07 如图 6-5 所示，图上的参数的含义如下：

- name：驱动程序的名称；
- Example URL：jdbc:phoenix: master;
- java Class path：要选中 phoenix 客户端的 jar 包；
- Class Name ：org.apache.phoenix.jdbc.PhoenixDriver。

配置完成后，单击 OK 按钮即可。

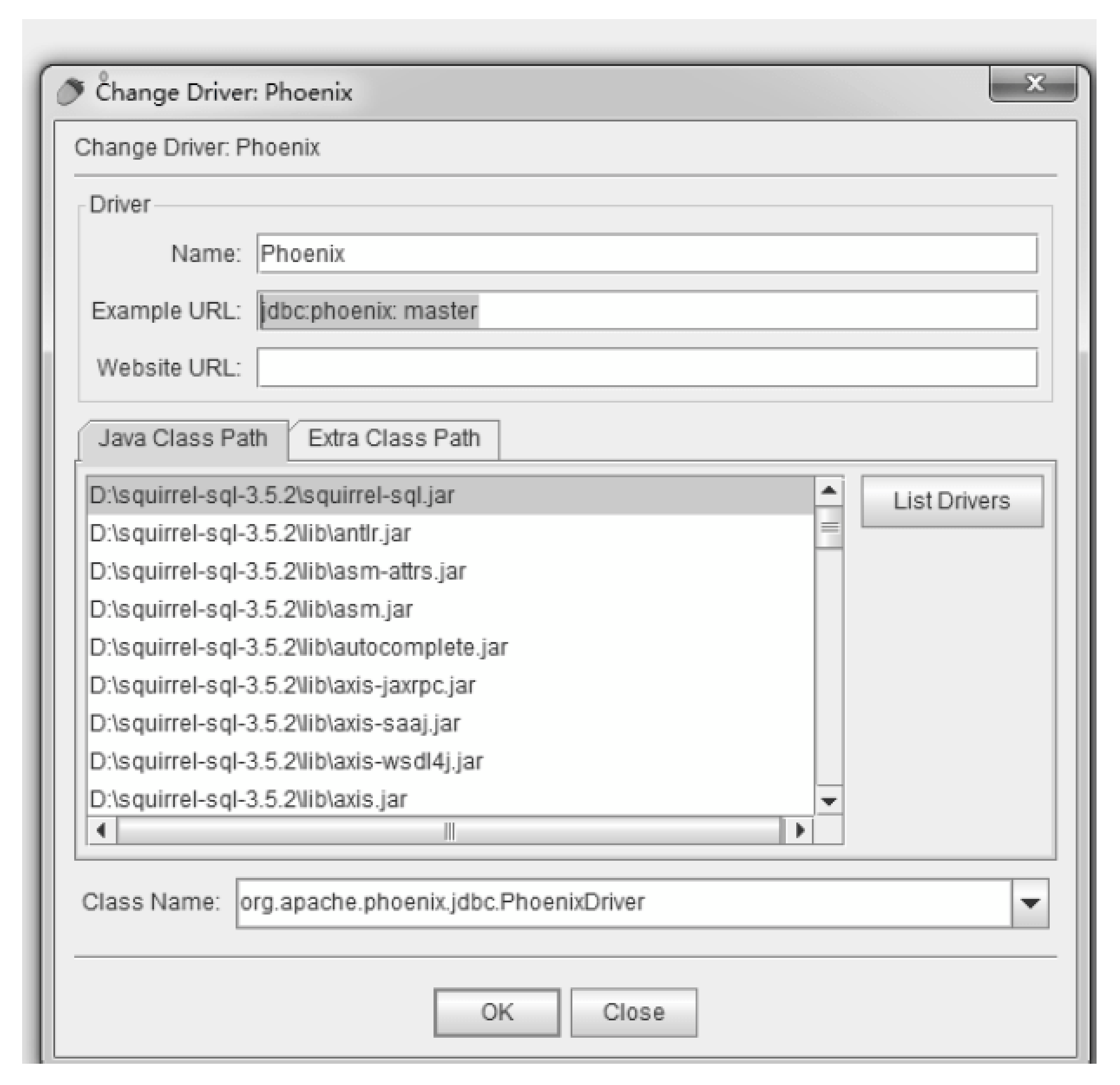


图 6-5　驱动程序配置

步骤 08　单击图 6-4 上的左侧的 Aliases，添加一个数据库连接的别名，具体参数如下（如图 6-6 所示）。

- Name：别名；
- Driver：选择我们刚刚添加的一个名叫 Phoenix 的 Driver;
- URL：输入在 Driver 里面的 Example URL 的内容;
- User Name:连接 HBase 的用户名;
- Password：上述用户名的密码。

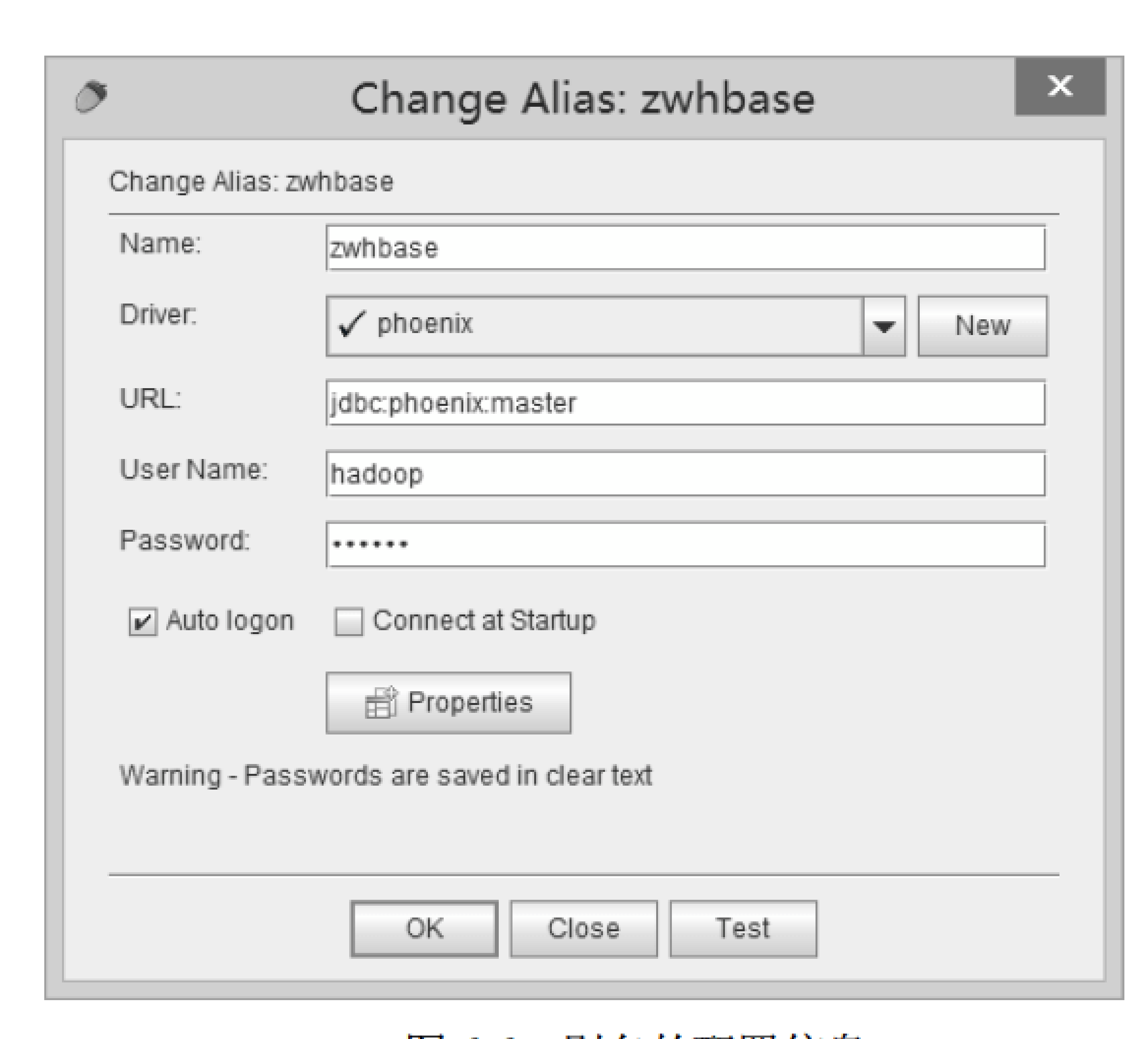


图 6-6　别名的配置信息



如果在 URL 那里填写的是 matser，那就必须在 Windows 的系统文件 hosts（在我机器上的位置是 C:\Windows\System32\drivers\etc）中配置一下 master 所对应的 IP 地址。我们可以在命令行窗口上来执行“ping master”。如果 ping 得通，说明配置正确。当然，你也可以直接使用 IP 地址。

步骤 09 右击刚刚创建的别名，在弹出菜单上选择 Connect，如图 6-7 所示。这时弹出如图 6-8 所示的窗口。

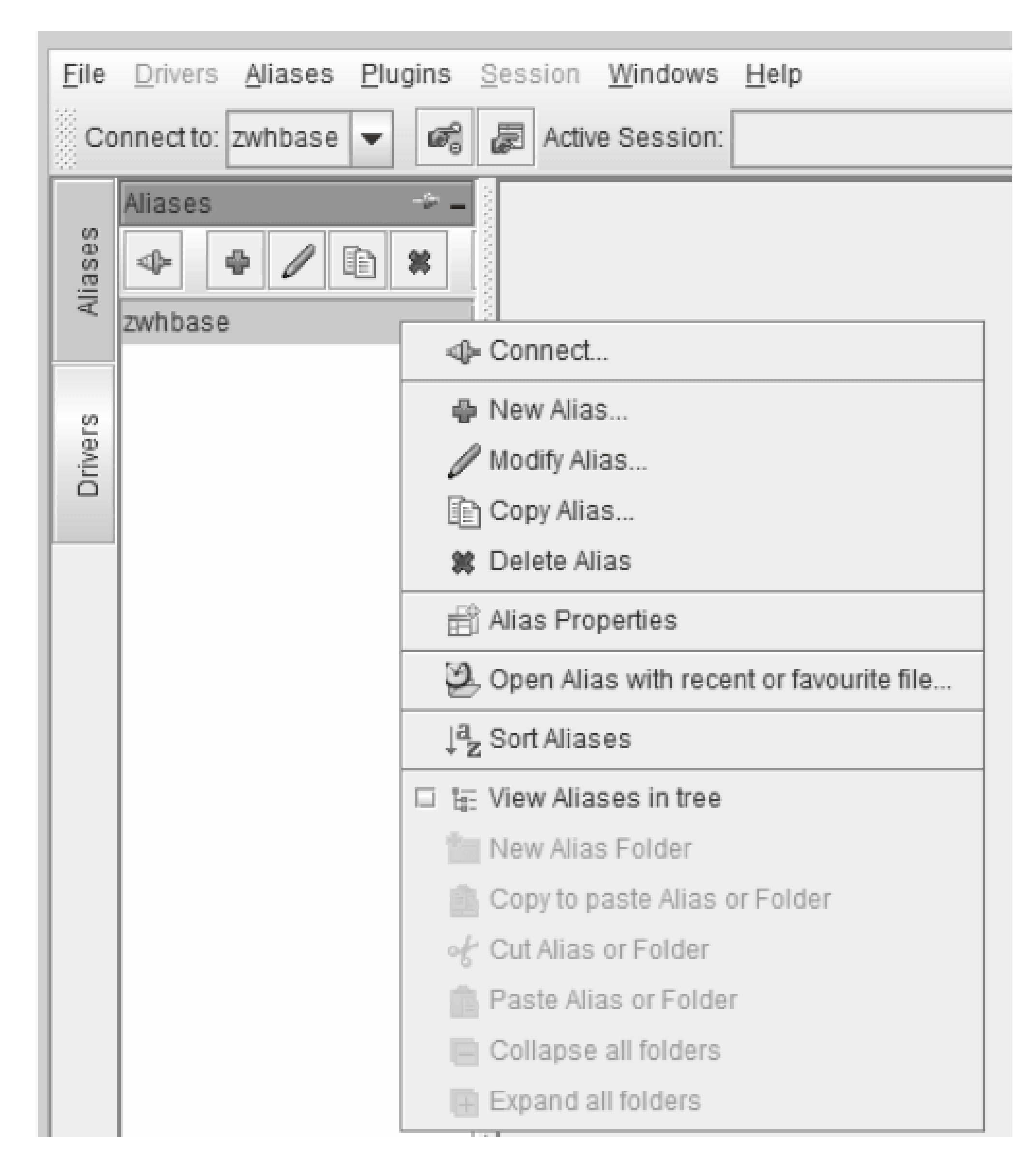


图 6-7　别名的菜单项



图 6-8　连接到 HBase 数据库

步骤 10　如果 B 连接成功，说明配置成功。之后就可以进行 SQL 操作了。如图 6-9 所示执行了一个 SELECT 查询。

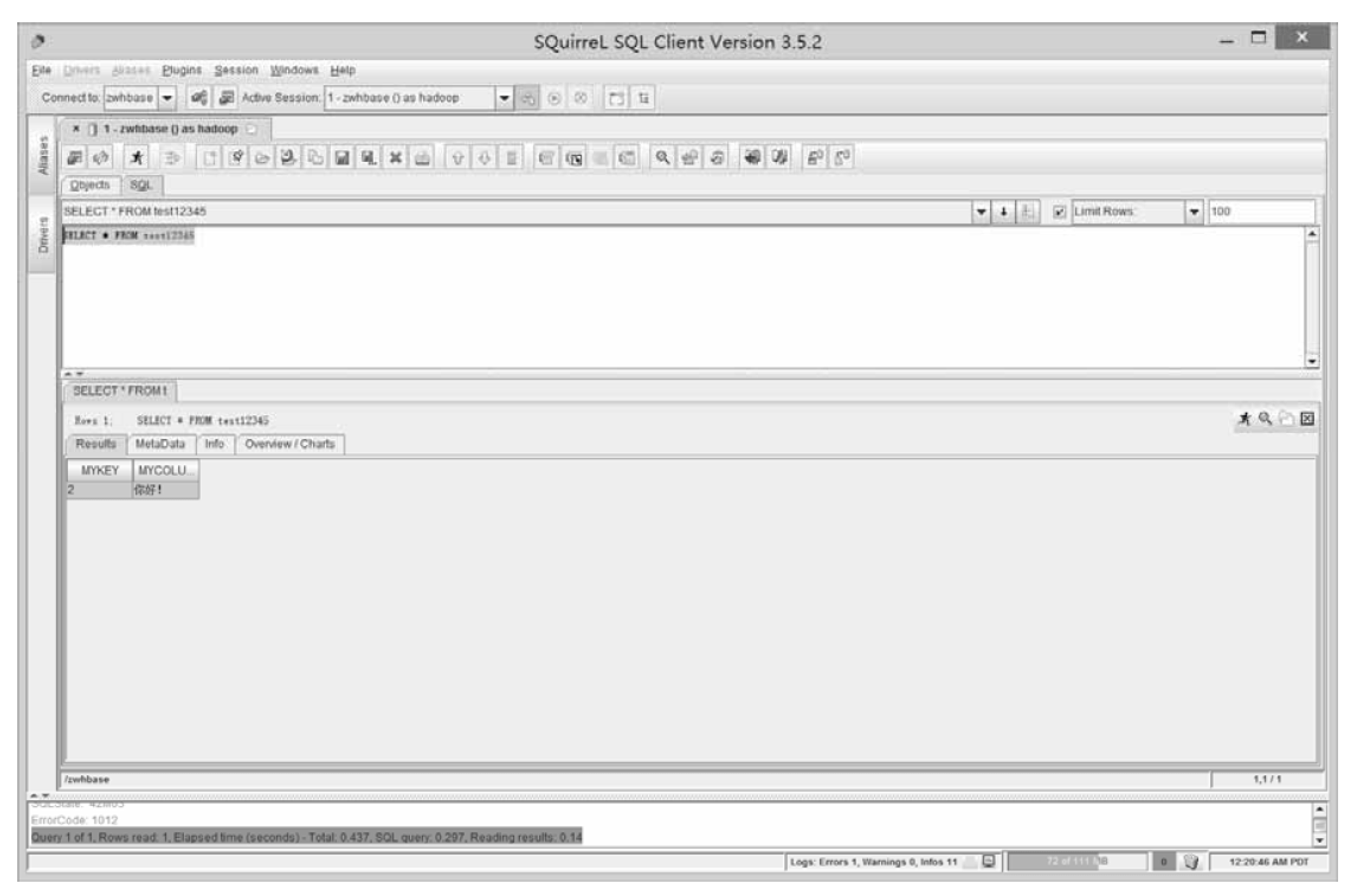


图 6-9　执行数据库查询

步骤 11　如果连接成功，可在 Objects 面板下看到如图 6-10 所示的数据库表等信息。

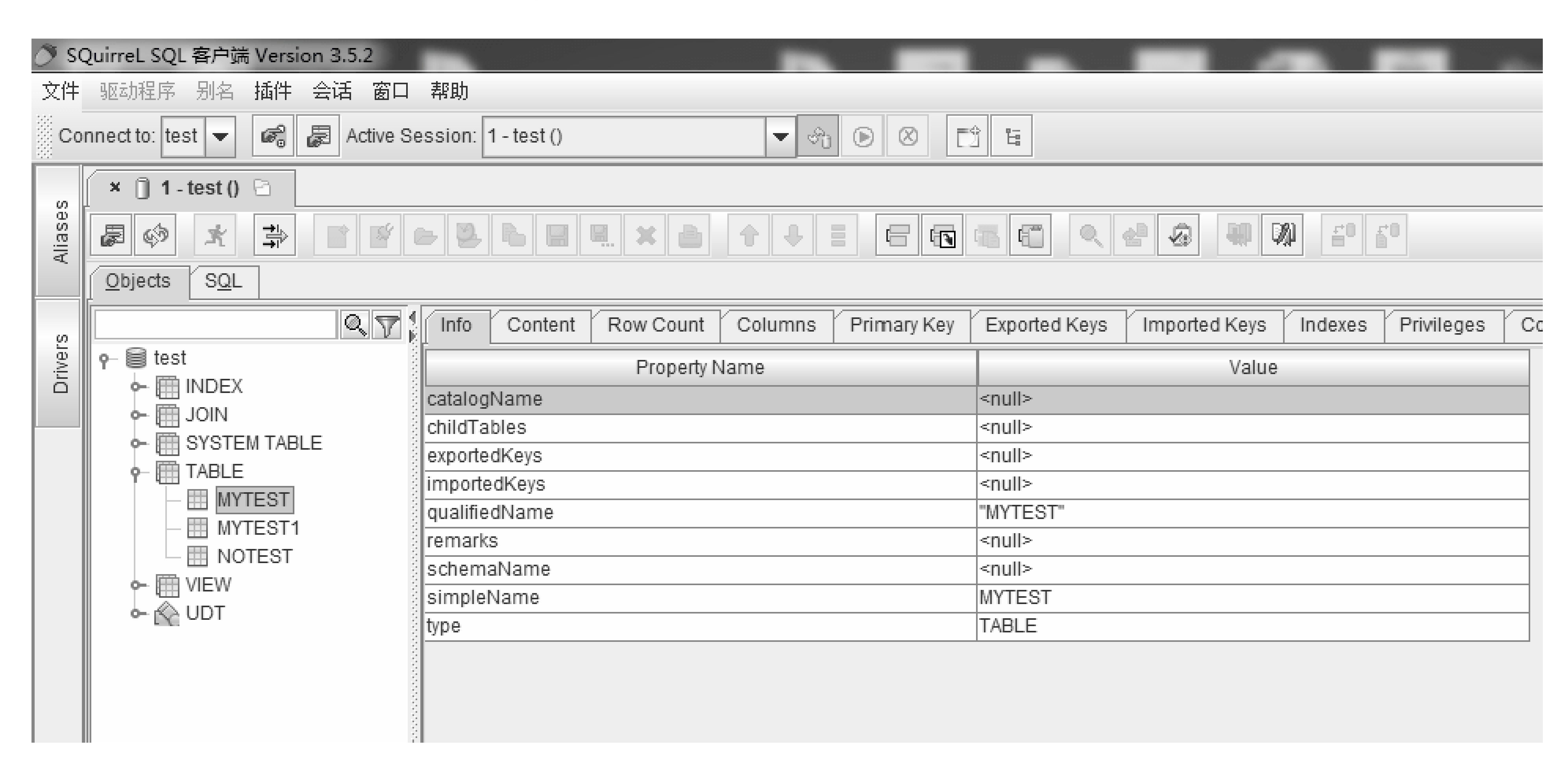


图 6-10　数据库表信息

步骤 12　在安装完成后，你可以执行数据库脚本（如果有的话）来创建你的数据库。可将整个脚本的 SQL 语句复制到 squirrel-sql 客户端的 SQL 执行界面，并执行之。

步骤13 查看刚创建的表，如图 6-11 所示。

```
DOCUMENTS
EMAILRULE
FILERULE
FOLDERLINK
IDOCUMENT
MAXIDFORNAME
MIMETYPES
MYSPACE
NAMECATALOG
NODETYPE
NODETYPEVIEW
PERMISSIONS
PERMISSIONSINROLE
PRINCIPALS
PROPERTYDEFINITION
PROPERTYMAP
PROPERTYREFERENCE
PROPERTYTYPE
PROPERTYTYPEGROUP
REFERENCEMINVALUE
REMOTESERVERS
REPLICATIONRULE
REPORTRULE
RULEONFOLDER
RULEONMESSAGE
RULEONSOURCE
SAVEDSEARCH
SECURITYROLES
STORAGES
SYSTEM.CATALOG
SYSTEM.SEQUENCE
TARGETCONTENTTYPES
TARGETSERVERS
TASKDEFINITION
USERGROUPS
WORKQUEUES
WQINROLE
YOUNGCOLLECTOR
YOUNGDRIVER
YOUNGSCHEDULE
61 row(s) in 0.5400 seconds

=> ["ACCESSCONTROLLISTS", "ACLENTRY", "ACTIVITIES", "BPMCONDITIONS", "BPMSTEPS",
 "BPMWORKS", "BUSINESSPROCESS", "BUSINESSROLE", "CHECKEDOUTCONTENT", "COLLECTRUL
E", "CONDITIONLISTS", "CONTAINERS", "CONTENTCATALOG", "CONTENTLINK", "CONTENTREF
ERENCE", "CONTENTTYPE", "CONTENTTYPELINK", "CONTENTTYPEVIEW", "DATABASERULE", "D
ELETEDDOCUMENTS", "DEVICERULE", "DOCUMENTS", "EMAILRULE", "FILERULE", "FOLDERLIN
K", "IDOCUMENT", "MAXIDFORNAME", "MIMETYPES", "MYSPACE", "NAMECATALOG", "NODETYP
E", "NODETYPEVIEW", "PERMISSIONS", "PERMISSIONSINROLE", "PRINCIPALS", "PROPERTYD
EFINITION", "PROPERTYMAP", "PROPERTYREFERENCE", "PROPERTYTYPE", "PROPERTYTYPEGRO
UP", "REFERENCEMINVALUE", "REMOTESERVERS", "REPLICATIONRULE", "REPORTRULE", "RUL
EONFOLDER", "RULEONMESSAGE", "RULEONSOURCE", "SAVEDSEARCH", "SECURITYROLES", "ST
ORAGES", "SYSTEM.CATALOG", "SYSTEM.SEQUENCE", "TARGETCONTENTTYPES", "TARGETSERVE
RS", "TASKDEFINITION", "USERGROUPS", "WORKQUEUES", "WQINROLE", "YOUNGCOLLECTOR",
 "YOUNGDRIVER", "YOUNGSCHEDULE"]
hbase(main):008:0>
```

图 6-11　查看 HBase 表

6.1.2　在 eclipse 上开发 phoenix 程序

在 eclipse 上配置 Phoenix 之前，我们首先确认 squirrel-sql-3.5.2 已经安装成功，并且可以连接上 master 的那台服务器。然后在机器上安装 eclipse。



要检查机器的操作系统是多少位的。如果是 32 位的，你的机器必须安装 32 位的 JDK 和 32 位的 eclipse；如果是 64 位的，你的机器必须安装 64 位的 JDK 和 64 位的 eclipse。

（1）在安装 eclipse 成功后，把 phoenix 的客户端 jar 包（phoenix-4.1.0-incubating-SNAPSHOT -client.jar 和 phoenix-core-4.1.0-incubating-SNAPSHOT.jar）拷贝到 eclipse 的安装目录的 plugins 文件夹下。

（2）启动 eclipse，新建一个 Java 项目，名称是 Phoenix，如图 6-12 所示。

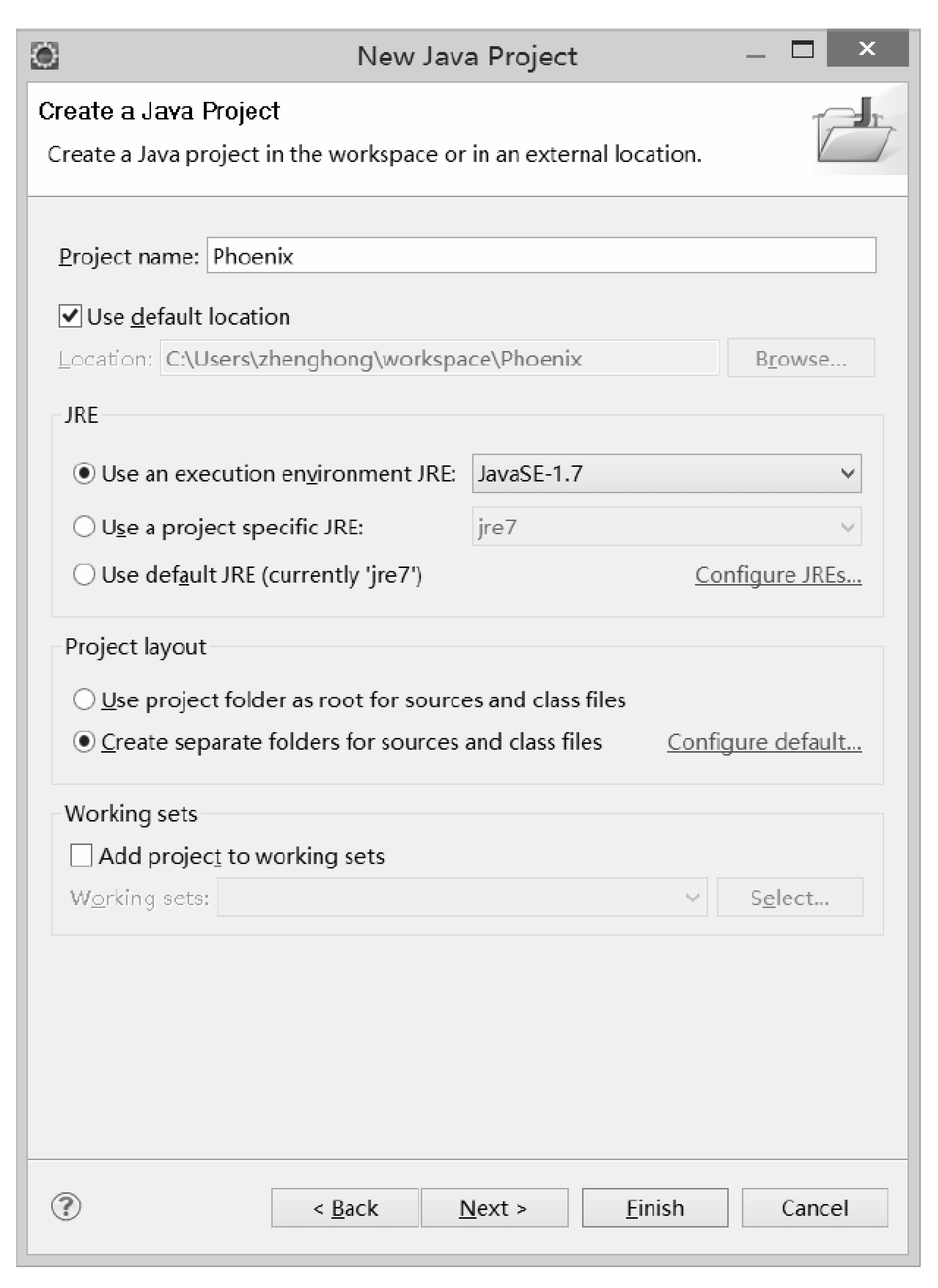


图 6-12 创建一个项目

（3）新建一个 Test 类。把 phoenix 的客户端的 jar 包添加到本项目下，步骤如下：

步骤 01 选中新建的项目，点击右键，选择 Build Path，选择 Configure Build Path，出现如图 6-13 所示的窗口。

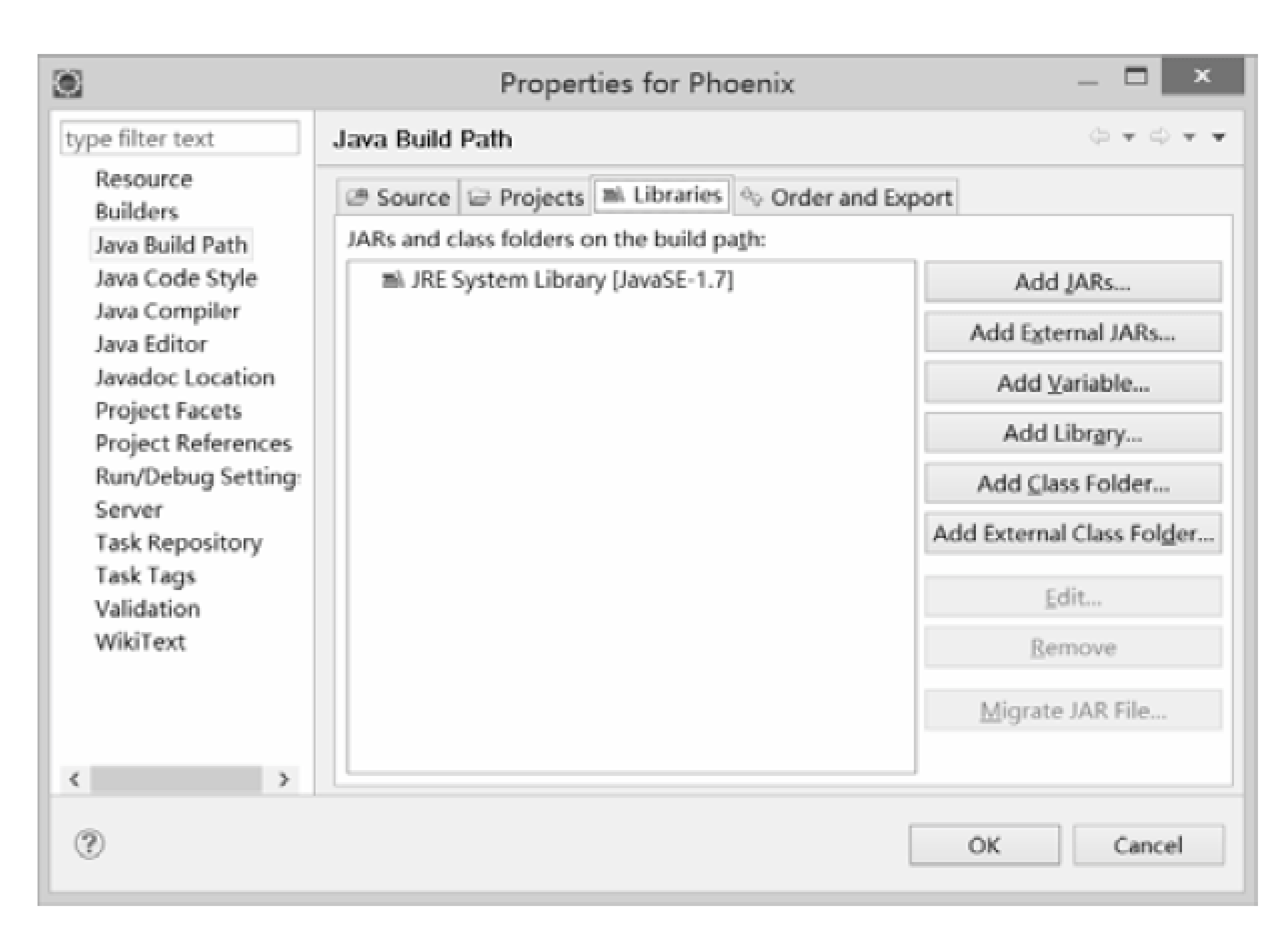


图 6-13　配置库文件

步骤 02 点击右侧的 Add External JARs，在弹出窗口中，选择 eclipse 安装文件的 plugin 的文件夹，选择刚刚拷贝到此文件夹下的 phoenix-4.1.0-incubating-SNAPSHOT-client.jar 和 phoenix-core-4.1.0-incubating-SNAPSHOT.jar，然后点击 OK，这个 phoenix 的客户端的 Jar 包就被添加到此项目下了，如图 6-14 所示。

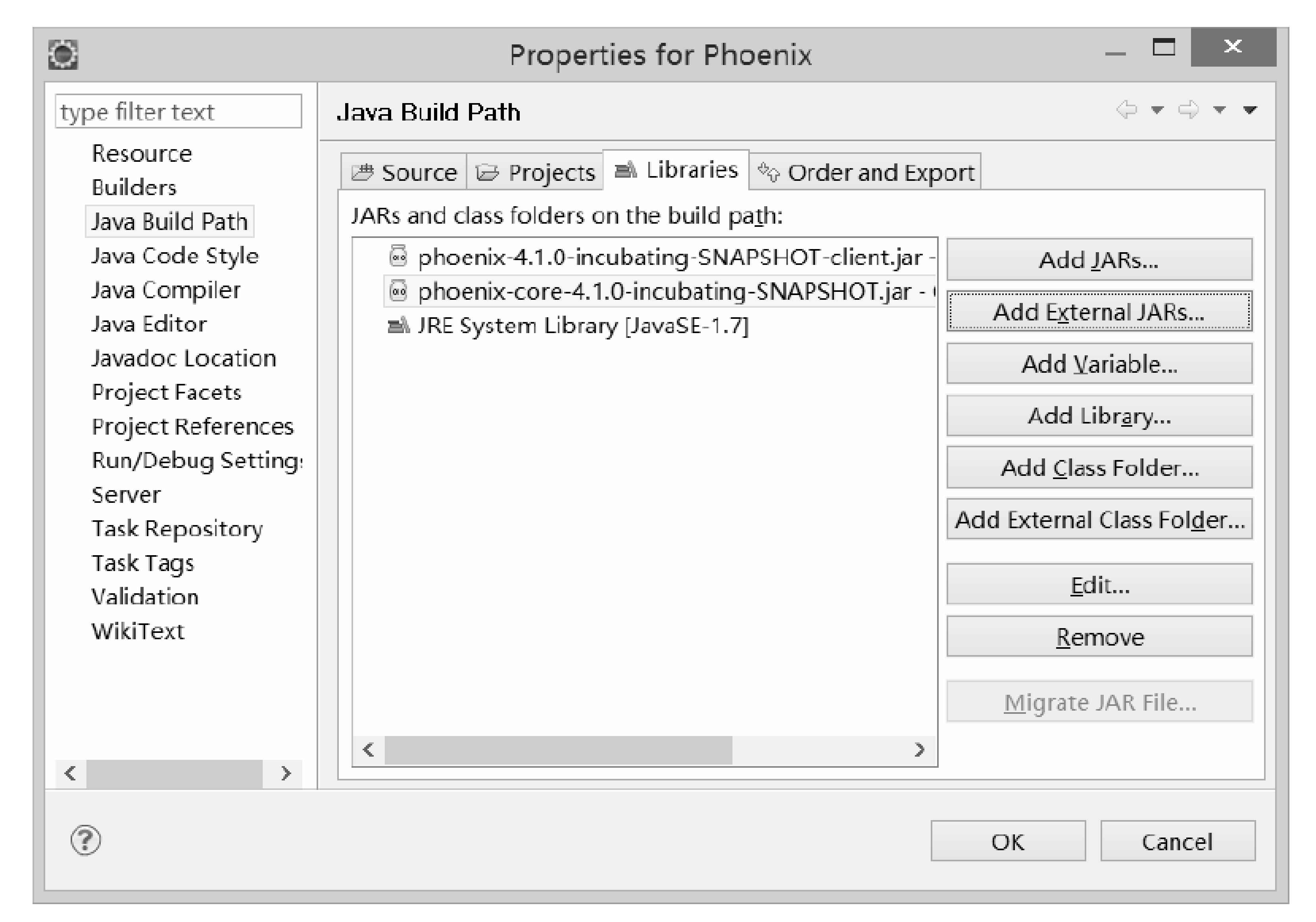


图 6-14　添加外部库

（4）回到主窗口，如图 6-15 所示。在 Test.java 中写代码，具体代码如下所示：

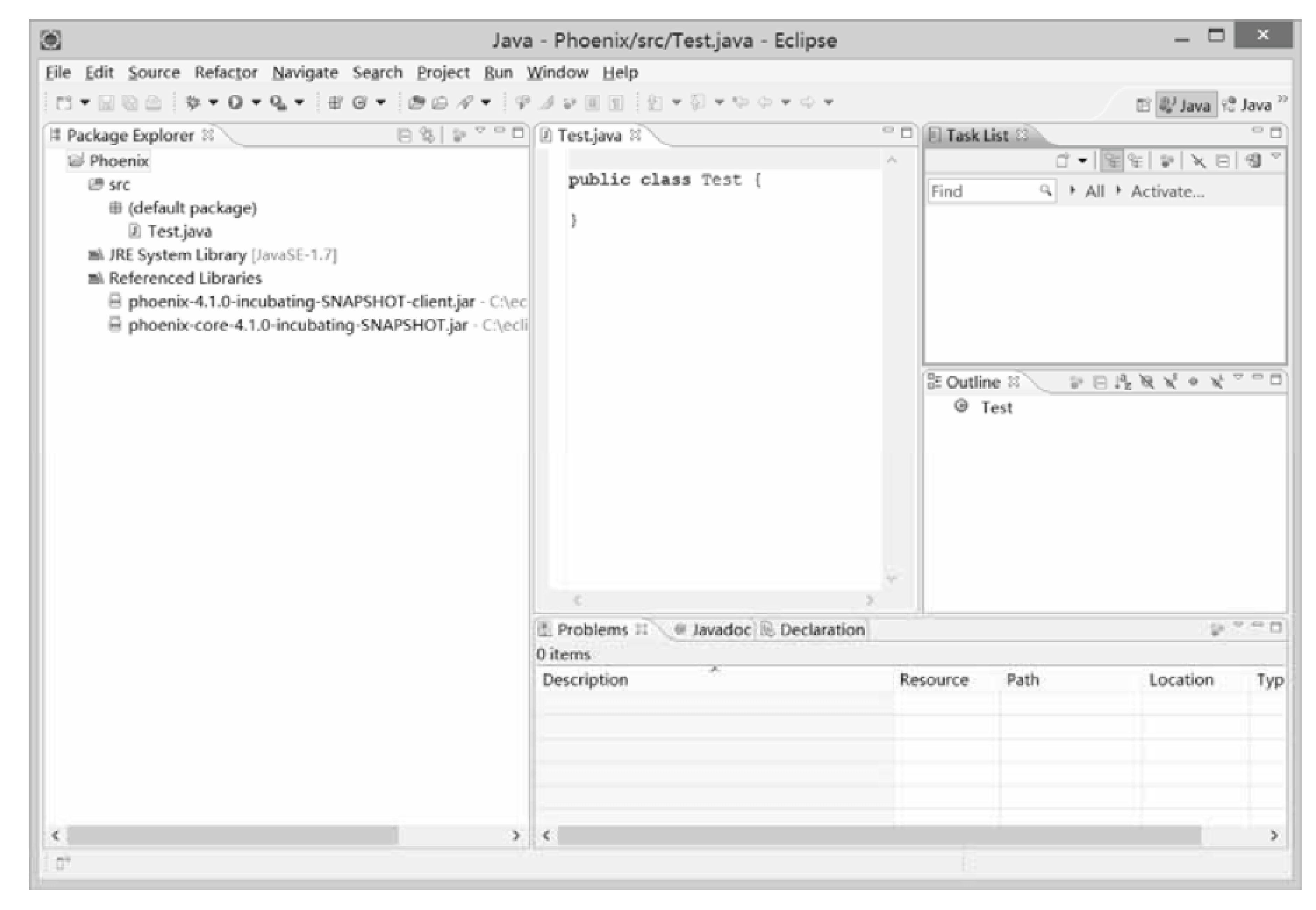


图 6-15　编写代码

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;
import java.sql.Statement;
public class Test {
   public static void main(String[] args) throws SQLException {
    Statement stmt = null;
    ResultSet rset = null;
    System.out.println("-----Connecting to hbase ------");
   Connection con = DriverManager.getConnection("jdbc:phoenix:master");
    stmt = con.createStatement();
    System.out.println("-----create table -----------");
    //新建表
    stmt.executeUpdate("create table test12345 (mykey integer not null
primary key, mycolumn varchar)");
    System.out.println("-----insert data -----------");
    stmt.executeUpdate("upsert  into  test12345  values  (1,'Hello')");
//插入数据
    stmt.executeUpdate("upsert  into  test12345  values  (2,'World1')");
//插入数据
    con.commit();     //提交操作
    //单个字段的查询
    //PreparedStatement statement = con.prepareStatement("select * from
test12345 where mykey=1");
    //......省略部分代码
    PreparedStatement statement;
    System.out.println("-----delete data -----------");
    stmt.executeUpdate("DELETE FROM TEST12345 WHERE mykey=1");
        //插入数据   （这个语句对表名的小写不敏感，可以大写，也可以小写）
        //phoenix 中没有修改的语句，只有覆盖，如果想修改的话，只要进行覆盖即可。
    System.out.println("-----update data -----------");
    stmt.executeUpdate("upsert into test12345 values (2,'你好！')");
    con.commit();
    //全表的查询
    System.out.println("-----query data -----------");
    statement  =  con.prepareStatement("select  *  from  test12345");
```

```
        rset = statement.executeQuery();
        while (rset.next()) {
            System.out.println(rset.getString("mycolumn"));
        }
        statement.close();
        con.close();
      }
    }
```

（5）运行 Test 程序。在这个程序运行成功后，你可以打开 phoenix 的可视化软件，查看 HBase 里面的 TEST12345 表的信息，如图 6-16 所示。到此为止，一个简单的 Phoenix 程序就算开发成功了。正如上面例子中所展现的，我们完全可以采用传统的 JDBC API 的形式编写代码。

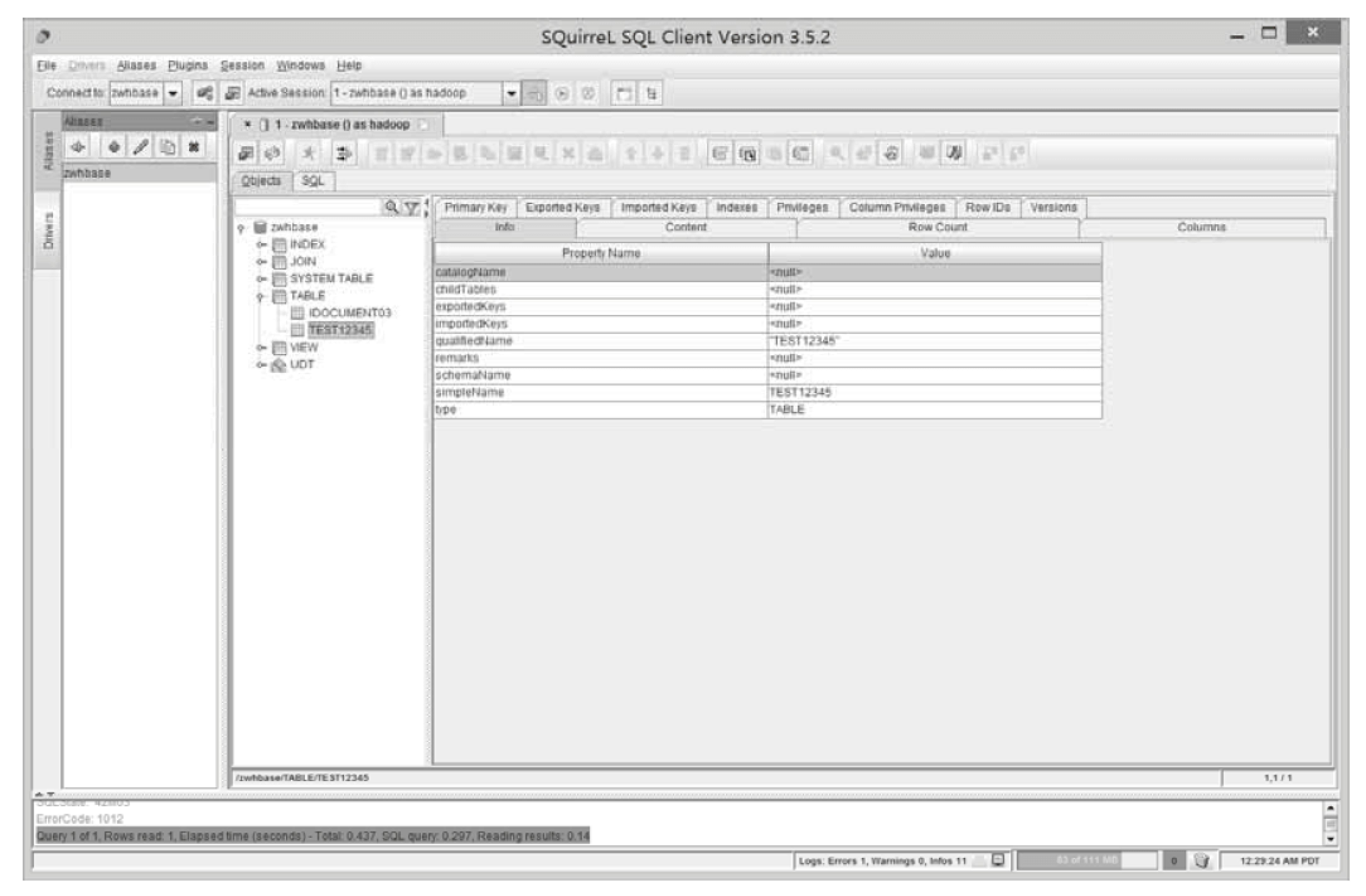


图 6-16 查看表信息

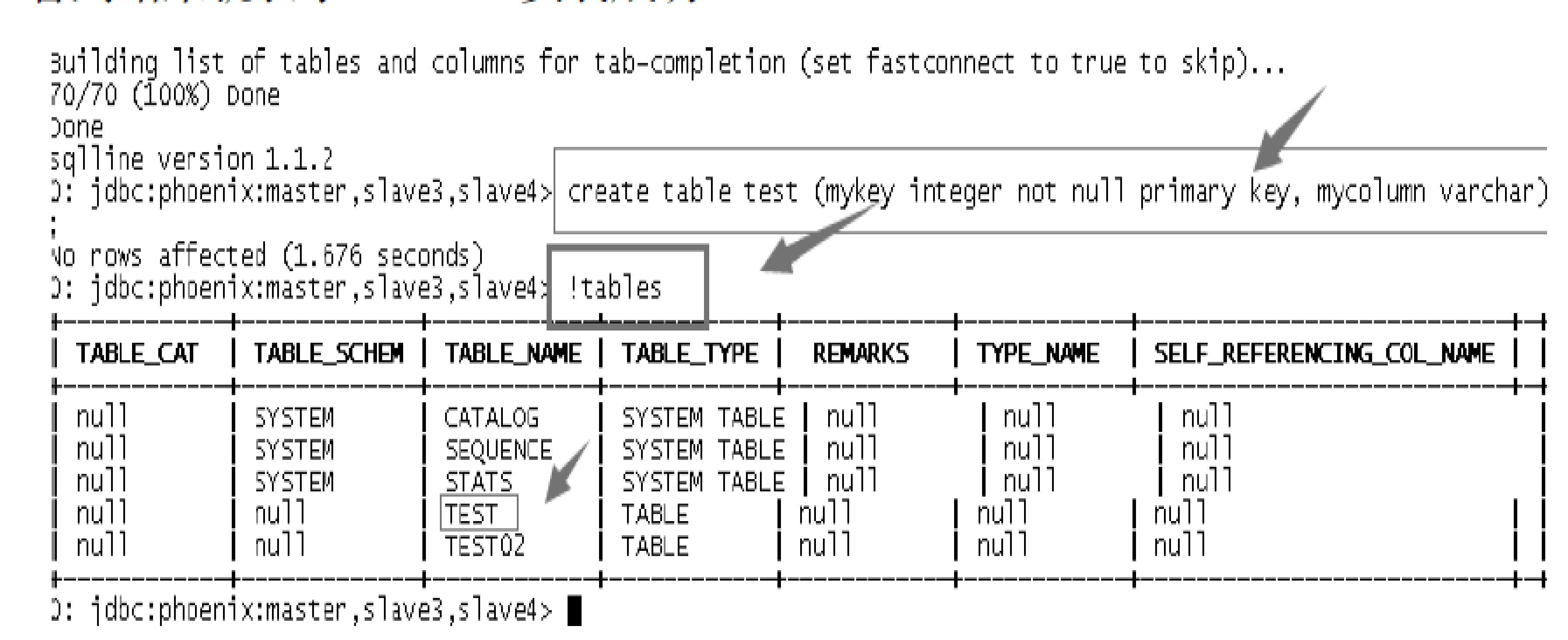
6.1.3 Phoenix SQL 工具

Phoenix 还支持其他的 SQL 命令行工具，比如：SQLLine。在 Linux 的 master 节点中，进入 Phoenix 的 bin 目录下，按照如图 6-17 所示，执行 sqlline.py 命令。

```
end2endTest.py  log4j.properties  performance.py  phoenix_utils.py  psql.py  sqlline.py
[root@master bin]# ./sqlline.py master,slave3,slave4
Setting property: [isolation, TRANSACTION_READ_COMMITTED]
issuing: !connect jdbc:phoenix:master,slave3,slave4 none none org.apache.phoenix.jdbc.PhoenixDriver
Connecting to jdbc:phoenix:master,slave3,slave4
15/01/22 18:24:52 WARN impl.MetricsConfig: Cannot locate configuration: tried hadoop-metrics2-phoenix.proper
es,hadoop-metrics2.properties
Connected to: Phoenix (version 4.2)
Driver: PhoenixEmbeddedDriver (version 4.2)
Autocommit status: true
Transaction isolation: TRANSACTION_READ_COMMITTED
Building list of tables and columns for tab-completion (set fastconnect to true to skip)...
70/70 (100%) Done
Done
sqlline version 1.1.2
0: jdbc:phoenix:master,slave3,slave4>
```

图 6-17　SQLLine 工具

输入一个 SQL 建表语句，之后输入“! tables”命令（该命令列出所有表的信息），如图 6-18 所示。看到结果就表示 Phoenix 安装成功。



```
Building list of tables and columns for tab-completion (set fastconnect to true to skip)...
70/70 (100%) Done
Done
sqlline version 1.1.2
0: jdbc:phoenix:master,slave3,slave4> create table test (mykey integer not null primary key, mycolumn varchar)
;
No rows affected (1.676 seconds)
0: jdbc:phoenix:master,slave3,slave4> !tables
```

TABLE_CAT	TABLE_SCHEM	TABLE_NAME	TABLE_TYPE	REMARKS	TYPE_NAME	SELF_REFERENCING_COL_NAME
null	SYSTEM	CATALOG	SYSTEM TABLE	null	null	null
null	SYSTEM	SEQUENCE	SYSTEM TABLE	null	null	null
null	SYSTEM	STATS	SYSTEM TABLE	null	null	null
null	null	TEST	TABLE	null	null	null
null	null	TEST02	TABLE	null	null	null

```
0: jdbc:phoenix:master,slave3,slave4>
```

图 6-18　创建表和显示表信息

在 SQLLine 下，你可以执行 describe 命令来查看表的信息，执行 SQL 脚本文件来批量加载数据，执行 SELECT 查询语句等等。

6.1.4　Phoenix SQL 语法

下面我们通过一些示例给出 Phoenix SQL 语法，用于通过 Phoenix 建立数据表、修改表、添加数据、修改数据、删除数据、删除表等操作。

- 新建一张 Person 表，含有 IDCardNum，Name，Age 三个字段，test 为表的 schema，SQL 语句如下：

```
create table IF NOT EXISTS test.Person (IDCardNum INTEGER not null primary key, Name varchar(20),Age INTEGER);
```

- 查询这个新表的数据，验证表的结构：

```
select * from TEST.PERSON;
```

- 对表进行插入操作:

```
upsert into Person (IDCardNum,Name,Age) values (100,'张三',18);
upsert into Person (IDCardNum,Name,Age) values (101,'李四',20);
upsert into Person (IDCardNum,Name,Age) values (103,'王五',22);
```

注意：在 Phoenix 中插入的语句为 upsert ，而不是 insert。

- 对表添加一列 sex（性别）:

```
ALTER TABLE test.Person ADD sex varchar(10);
select * from test.person;
```

后一个 SELECT 语句是验证我们已经新增了列 sex，每行的默认值为 null 。

- 更新表数据:

Phoenix 中不存在 update 的语法关键字，而是 upsert ，功能上替代了 Insert+update，如果该行存在，则更新，否则就插入。语句如下：

```
upsert into test.person (idcardnum,sex) values (100,'男');
upsert into test.person (idcardnum,sex) values (101,'女');
upsert into test.person (idcardnum,sex) values (103,'男');
select * from test.person;
```

- 复杂查询，支持 where、group by、case when 等复杂的查询条件:

```
upsert into test.Person (IDCardNum,Name,Age,sex) values (104,'小张',24,'男');
upsert into test.Person (IDCardNum,Name,Age,sex) values (105,'小李',26,'男');
upsert into test.Person (IDCardNum,Name,Age,sex) values (106,'小李',28,'男');
select * from test.person;
select sex ,count(sex) as num from test.person where age >20 group by sex;
```

- 删除数据及删除表，标准 SQL 如下:

```
delete from test.Person where idcardnum=100;
select * from test.Person where idcardnum=100;
drop table test.person;
!tables
```

6.2 Hive

Hive 是一种建立在 Hadoop 之上的数据仓库架构。它提供了一种让用户对数据描述其结构的机制，支持用户对存储在 Hadoop 中的海量数据进行查询和分析的能力。Hive 简单，容易上手，它提供了类 SQL 查询语言 HQL。Hive 提供了统一的元数据管理。

6.2.1　Hive 架构

Facebook 在 2010 ICDE 会议上介绍了数据仓库 Hive。Hive 存储海量数据在 Hadoop 系统中，提供了一套类数据库的数据存储和处理机制。它采用类 SQL 语言对数据进行自动化管理和处理，经过语句解析和转换，最终生成基于 Hadoop 的 MapReduce 任务，通过执行这些任务完成数据处理（Hive 也可以基于 Spark）。

Hive 是一个基于 Hadoop 的数据仓库工具。Hive 查询有一定的延时，常被用来进行静态数据分析和挖掘。图 6-19 所示显示了 Hive 的主要组件。ODBC 和 JDBC 是编程接口，驱动器对输入进行编译、优化和执行。MetaStore（元数据存储）是一个独立的 RDBMS，默认是内置的 Apache Derby 数据库。对于生产系统，推荐使用 MySQL 或其他 RDBMS。Hive 会在其中保存表模式和其他系统元数据。

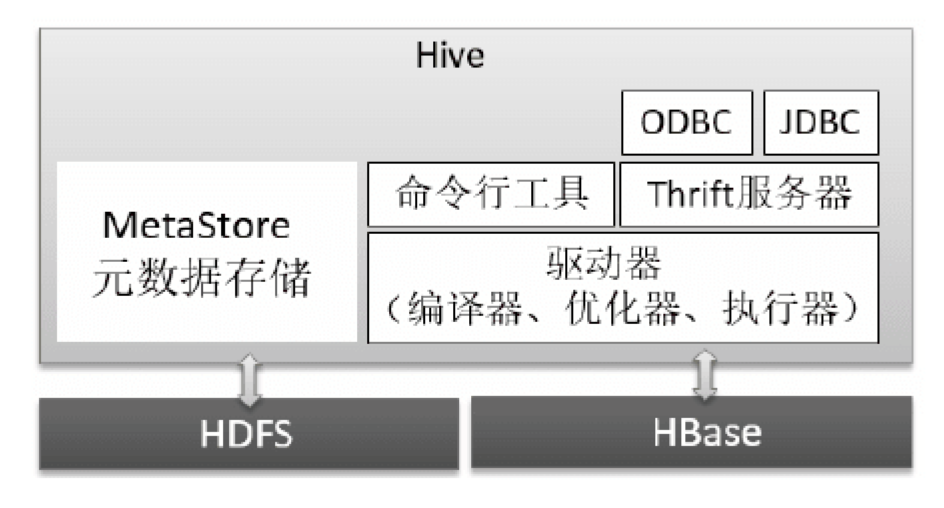


图 6-19　Hive 架构

Hive 不支持行级别的更新，不支持实时的查询响应速度。如果需要这些功能，在底层可以使用 HBase（HBase 并没有提供类似 SQL 的查询语言，所以 Hive 可以和 HBase 结合使用）。因为大多数数据仓库应用程序都是基于 SQL 的 RDBMS 实现的，所以 Hive 降低了将这些应用程序移植到 Hadoop 上的困难，减少了开发人员的学习成本。用户只要熟悉 SQL，那么，使用 HiveQL 就会很容易。Hive 还支持用户自定义函数，用户可以根据自己的需求来实现自己的函数。

Hive 提供了数据的查询、分析和聚集。需要注意的是：Hive 没有数据的插入、删除和更新，数据进入 Hive 是通过装载工具完成的。通过 Hive，可以将在 HDFS 上的结构化的数据文件映射为一张数据库表。Hive 定义了一个类似于 SQL 的查询语言 HiveQL，可以实现复杂查询和 Join，能够将用户编写的 SQL 转化为相应的 MapReduce 程序，并最终在 Hadoop 上执行。还有，HiveQL 支持在查询中嵌入 MapReduce 脚本。Hive 并不是用来操作 OLTP 的需求，它不提供

实时查询或行级的更新。它非常适合于对那些只在文件末尾添加数据的大型数据集（如：Web 日志）进行批处理。Hive 支持文本文件 TextFile、SequenceFiles（包含二进制键/值对的文本文件）和 RCFiles（Record Columnar Files，采用列数据库的模式存储一个表的列）。前两个文件格式都属于行存储方式，后一个是列式存储，能更快地进行数据装载和查询。这对于数据仓库而言是非常关键的。比如：每天大约有超过 20TB 的数据上传到 Facebook 的数据仓库，由于数据加载期间网络和磁盘流量会干扰正常的查询执行，因此缩短数据加载时间是非常有必要的。另外，为了满足实时性的网站请求和支持高并发用户提交查询的大量读负载，查询响应时间也是非常关键的，这就要求底层存储结构能够随着查询数量的增加而保持高速的查询处理。

6.2.2 安装 Hive

用户可以通过多种方式来安装 Hive。一种方式是从 Hive 的官方网站 http://hive.apache.org/下载一个 Hive 软件压缩包。然后，进行解压 Hive，并添加 Hive 环境变量。

Hive 主要包含三个部分。在$HIVE_HOME/lib 目录下有很多 JAR 文件，每个 JAR 文件都实现了 Hive 功能中某个特定的部分。在$HIVE_HOME/bin 目录下包含了可以执行 Hive 服务的可执行文件，其中包括了 Hive CLI。CLI 提供交互式的界面输入语句或脚本。在$HIVE_HOME/conf 目录下存放了 Hive 的配置文件。

下面进入 conf 目录，配置 Hive。依据 hive-env.sh.template，创建 hive-env.sh 文件，这个文件是 Hive 运行环境的配置文件：

```
cp hive-env.sh.template hive-env.sh
```

然后修改 hive-env.sh，指定 hive 配置文件的路径，指定 Hadoop 路径：

```
export HIVE_CONF_DIR=/home/test/Desktop/hive/conf
HADOOP_HOME=/home/test/Desktop/hadoop
```

接下来启动 Hive。在 CLI 命令行键入 hive，则显示：

```
   WARNING: org.apache.hadoop.metrics.jvm.EventCounter is deprecated.
Please use org.apache.hadoop.log.metrics.EventCounter in all the
log4j.properties files.
   Logging initialized using configuration in
jar:file:/home/test/Desktop/hive-0.8.1/lib/hive-common-0.8.1.jar!/hive-
log4j.properties
   Hive history
file=/tmp/test/hive_job_log_test_201208260529_167273830.txt
   hive>
```

输入一些测试语句，比如：建立测试表 test：

```
create table test (key string);
show tables;
show databases;
```

Hive 有一个默认的数据库，叫做 default。如果没有指定数据库，则就会使用这个 default 数据库。show databases 命令就可以显示 Hive 上所包含的数据库。最后，我们看一下 hive-site.xml 文件，这是 Hive 的配置文件。用户所做的配置修改只需要在这个文件中进行即可。

```
......
<property>
<name>javax.jdo.option.ConnectionURL</name>
<value>jdbc:derby:;databaseName=metastore_db;create=true</value>
<description>JDBC connect string for a JDBC metastore</description>
</property>
<property>
<name>javax.jdo.option.ConnectionDriverName</name>
<value>org.apache.derby.jdbc.EmbeddedDriver</value>
<description>Driver class name for a JDBC metastore</description>
</property>
<property>
<name>javax.jdo.option.ConnectionUserName</name>
<value>APP</value>
<description>username to use against metastore database</description>
</property>
<property>
<name>javax.jdo.option.ConnectionPassword</name>
<value>mine</value>
<description>password to use against metastore database</description>
</property>
......
```

在上面的文件中，javax.jdo.option.ConnectionURL 告诉 Hive 如何连接 metastore 服务器。databaseName 就是数据库名称。正如上面提到的，metastore 是元数据存储组件，存储了表的模式和分区信息等元数据信息。用户在执行 create table 或者 alter table 语句时会指定这些信息。数据库类型默认为 Debry（Apache Derby 是一个完全用 Java 编写的数据库，所以可以跨平台，但需要在 JVM 中运行。Derby 是一个 Open Source 的产品，基于 Apache License 2.0 分发），即将元数据存储在 Derby 数据库中，也是 Hive 默认的安装方式。Debry 提供了有限的单进程的服务，所以 Debry 并不适合于生产环境。在实际项目中，我们一般采用 MySQL。

6.2.3 Hive 和 MySQL 的配置

为了让 Hive 使用 MySQL 作为元数据存储，我们首先安装 MySQL。如果你使用 Ubuntu，则用 apt-get 安装（sudo apt-get install mysql-server）。在安装之后，你创建数据库 hive，创建 Hive 用户，并授权：

```
create database hive
grant all on hive.* to hive@'%'  identified by 'hive';
flush privileges;
```

然后修改 hive-site.xml，指定 MySQL 的服务器和端口信息，指定数据库名称。比如：

```
<property>
<name>javax.jdo.option.ConnectionURL </name>
<value>jdbc:mysql://localhost:3306/hive </value>
</property>
<property>
<name>javax.jdo.option.ConnectionDriverName </name>
<value>com.mysql.jdbc.Driver </value>
</property>
<property>
<name>javax.jdo.option.ConnectionPassword </name>
<value>hive </value>
</property>
<property>
<name>hive.hwi.listen.port </name>
<value>9999 </value>
<description>This is the port the Hive Web Interface will listen on
</descript ion>
</property>
<property>
<name>datanucleus.autoCreateSchema </name>
<value>false </value>
</property>
<property>
<name>datanucleus.fixedDatastore </name>
<value>true </value>
</property>
<property>
<name>hive.metastore.local </name>
```

```
    <value>true </value>
    <description>controls whether to connect to remove metastore server or open a new metastore server in Hive Client JVM </description>
    </property>
```

我们还需要把 MySQL 的驱动程序存放在$HIVE_HOME/lib 目录下。在正确设置驱动和配置之后，Hive 就会把元数据放在 MySQL 上了。下面就可以启动 Hive，并建立测试表 test 了：

```
create table test (key string);
show tables;
```

6.2.4　Hive CLI

Hive 命令行工具（CLI）是和 Hive 交互的最常用的方式。下面我们通过几个例子来说明 CLI 的使用方式。首先我们创建一个普通的文本文件，里面只有一行数据，存储了一个字符串，命令如下：

```
echo 'yangzhenghong' > /home/hadoop/test.txt
```

然后我们建一张 hive 的表 test，其中-e 选项就是让 hive CLI 在执行命令后立即退出：

```
hive -e "create table test (value string)";
```

进入 hive，接下来加载数据：

```
Load data local inpath 'home/hadoop/test.txt' overwrite into table test
```

最后我们查询如下：

```
hive> select * from employee;
OK
yangzhenghong
Time taken: 0.566 seconds, Fetched: 1 row(s)
hive>
```

类似 RDBMS，Hive 还支持执行 SQL 脚本文件，比如：

```
hive -f /home/Hadoop/test.hql
```

在 Hive CLI 上可以直接执行 shell 命令，只需要在命令前面加上！即可：

```
hive > ! pwd;
```

6.2.5　Hive 数据类型

Hive 支持 RDBMS 中的大多数数据类型，比如：TINYINT、SMALLINT、INT、BIGINT、

BOOLEAN、FLOAT、DOUBLE、STRING、TIMESTAMP、BINARY 等。所有这些数据类型都是对 Java 中的接口的实现，比如：STRING 类型实现的是 Java 中的 String。与 RDBMS 不同的是，Hive 还支持集合数据类型，比如：STRUCT（与 C 语言的 struct 类似，可通过“点”名称来访问。比如：STRUCT (street STRING, city String)，那么， 列名.street 就是第一个元素）、MAP（一组键-值对的集合）、ARRAY（数组）。我们来看一个例子：

```
CREATE TABLE employees (
empID       INT,
 name        STRING,
 salary      FLOAT,
 skills          ARRAY<STRING>,
educations MAP<STRING, STRING>,
address     STRUCT<street:STRING, city:STRING>) ;
```

其中 empID 是员工编号，是一个整数。name 是姓名，是一个字符串。salary 是工资，是一个浮点数。skills 就是该员工的技能，是一个字符串数组(如果用 SELECT 语句查询这个列，则返回的数组会以[“java”,” C++”]的 JSON 格式显示)。educations 是该员工的教育背景，是一个由键-值对构成的 map（如果用 SELECT 语句查询这个列，返回的结果也是 JSON 格式，在{}之内以逗号分隔“键：值”对），记录了年份（用字符串表示）和毕业学校之间的对应关系。address 是地址，使用 STRUCT（如果使用 SELECT 查询，返回的结果也是 JSON，类似 MAP 的返回格式），每个域都有一个名字和类型。针对 SELECT 查询，我们给出以下几个例子：

```
SELECT name, skills[0] FROM employees;
SELECT name, educations[ “University” ] FROM employees;
SELECT name, address.city FROM employees;
```

正如上面所列出的，对于数组列，同 JAVA 一样，数组索引从 0 开始。对于 MAP 列，可以指定键值，而对于 STRUCT，可以用“点”符号。

Hive 可以把表的数据放到文本文件上，或者从文本文件上读取表的数据，所以，文件的格式化是需要的，即：怎么分隔行记录和列。我们所熟悉的，用逗号或者制表符分隔的模式，Hive 也支持。但是，Hive 默认使用几个控制符号，因为这些符号在列值中很少出现，如表 6-1 所示。

表 6-1　Hive 默认使用的控制符号

分隔符	说明
\n	对于文本文件来说，每行就是一条记录，因此换行符可以分割行记录
^A (Ctrl+A)	用于分隔列，在 CREATE TABLE 语句中可以使用八进制编码\001 表示
^B	用于分隔 ARRAY 或 STRUCT 中的元素，或用于 MAP 中键-值对之间的分隔，在 CREATE TABLE 语句中可以使用八进制编码\002 表示
^C	用于 MAP 中键和值之间的分隔，在 CREATE TABLE 语句中可以使用八进制编码\003 表示

在 CREATE TABLE 上可以指定分隔符，比如：

```
CREATE TABLE employees (
empID   INT,
name    STRING,
salary  FLOAT,
skills     ARRAY<STRING>,
educations MAP<STRING, STRING>,
address    STRUCT<street:STRING, city:STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\001'
COLLECTION ITEMS TERMINIATED BY '\002'
MAP KEYS TERMINATED BY '\003'
LINES TERMINATED BY '\n'
STORED AS TEXTFILE;
```

在上面的语句中，“FIELDS TERMINATED BY ‘\001’”指定了使用^A 作为列的分隔符，而“COLLECTION ITEMS TERMINIATED BY ‘\002’”指定了使用^B 作为集合元素之间的分隔符，“MAP KEYS TERMINATED BY ‘\003’”指定了使用^C 作为 MAP 的键和值之间的分隔符。虽然用户可以明确指定这些子句，但是，在大多数情况下，默认的分隔符就够用了。需要特别指出的是，如果不是文本格式，而是 CSV 格式（值是用逗号分隔的）和 TSV 格式（值是用制表键分隔的），这些格式的文件中文件头包含有列名，列值字符串是用引号包含的。

Hive 的这种文件读取数据的模式非常适合处理由 ETL 工具所产生的数据文件。那么，如果文件内容不匹配表的模式，Hive 会怎么办呢？如果文件中每行记录中的列（字段）个数少于对应的模式中定义的列个数，那么，Hive 结果中会有很多 null 值。如果列是数值型的，而文件中的数据时非数值型的，则 Hive 将处理为 null 值。

Hive 支持用户自定义函数（UDF），可以和内置的函数一样使用。SHOW FUNCTIONS 命令可以列出 Hive 中所有的函数名称，包括内置的和用户自定义的函数。DESCRIBE FUNCTION 命令可以显示函数的信息。下面来看一个 UDF 的例子。这个 UDF 的输入值是一个日期，输出结果是该日期所对应的星座。对于 UDF，我们首先编写相对应的 JAVA 代码，代码如下所示。

```
package org.apache.hadoop.hive.contrib.udf.example;
import java.util.Date;
import java.text.SimpleDateFormat;
import org.apache.hadoop.hive.ql.exec.UDF;
//函数注解信息，用于说明这个函数的使用方法
@Description(name = "zodiac",
  value = "_FUNC_(date) - from the input date string "+
            "or separate month and day arguments, returns the sign
```

```
of the Zodiac.";
      extended = "Example:\n"
        + " > SELECT _FUNC_(date_string) FROM src;\n"
        + " > SELECT _FUNC_(month, day) FROM src;")

   //UDF 函数所对应的 JAVA 类必须继承 UDF 类
   public class UDFZodiacSign extends UDF{
      private SimpleDateFormat df;
      public UDFZodiacSign(){
        df = new SimpleDateFormat("MM-dd-yyyy");
      }

      public String evaluate( Date bday ){
        return this.evaluate( bday.getMonth(), bday.getDay() );
      }

      public String evaluate(String bday){
        Date date = null;
        try {
           date = df.parse(bday);
        } catch (Exception ex) {
        return null;
      }
      return this.evaluate( date.getMonth()+1, date.getDay() );
   }
      public String evaluate( Integer month, Integer day ){
        if (month==1) {
           if (day < 20 ){
              return "Capricorn";
           } else {
              return "Aquarius";
           }
        }
        if (month==2){
           if (day < 19 ){
              return "Aquarius";
           } else {
              return "Pisces";
        }
```

```
        }
        /* 省略了其他月份的处理代码... */
        return null;
    }
}
```

将上述代码编译并打包到一个 JAR 文件中，并将这个 JAR 文件加入到类路径下。而后，定义这个 UDF 函数。最后就可以像使用内置函数一样使用它了。比如：

```
hive> ADD JAR /full/path/to/zodiac.jar;
hive> CREATE TEMPORARY FUNCTION zodiac
> AS 'org.apache.hadoop.hive.contrib.udf.example.UDFZodiacSign';
hive> SELECT name, bday, zodiac(bday) FROM testdata;
```

删除 UDF 的命令是 DROP TEMPORARY FUNCTION。除了继承 UDF 类来创建自定义的函数之外，我们也可以继承 GenericUDF 类。GenericUDF 能够支持更复杂的输入处理，具体内容，可参考 Hive 文档。

6.2.6　HiveQL DDL

除了 default 数据库之外，Hive 会为每个数据库创建一个目录，数据库中的表将会以这个数据库目录的子目录形式存储。比如：

```
CREATE DATABASE yunsheng
```

那么，Hive 将会创建一个目录/user/hive/warehouse/yunsheng.db。使用 DESCRIBE DATABASE yunsheng 语句就会显示这个数据库所在的目录信息。使用 USE yunsheng 命令就可以将 yunsheng 数据库设置为当前的工作数据库。然后，SHOW TABLES 就会显示这个数据库下的所有表。DROP DATABASE yunsheng 就可以删除数据库。默认情况下，Hive 不允许删除一个包含表的数据库，用户要么先删除表，然后再删除数据库，或者在删除命令的后面添加 CASCADE 关键字，这样 Hive 就会先删除表，然后再删除数据库了，比如“DROP DATABASE yunsheng CASCADE”。

在前面的章节中，我们使用了 CREATE TABLE 创建表。在创建表的时候，Hive 会自动增加两个表属性：一个是 last_modified_by（保存着最后修改这个表的用户名），一个是 last_modified_time（保存着最后修改这个表的时间）。在创建表时，我们可以为表指定存储路径，比如：

```
CREATE TABLE IF NOT EXISTS yunsheng.contracts(
 name STRING COMMENT 'contract name',
 ……)
 LOCATION '/user/hive/warehouse/yunsheng.db/contracts';
```

```
LOAD DATA INPATH '/tmp/result/20160413' INTO TABLE contracts;
```

在创建表之后，可以使用 SHOW TABLES 来列举所有的表。如果想要看某个列的信息，则可以使用 DESCRIBE yunsheng.contracts.name 命令。Hive 还支持创建视图。

Hive 表的管理方式有两种：Managed Tables（内部表） 和 External Tables（外部表）。上面所创建的表都是内部表。Hive 创建内部表时，会将数据移动到数据仓库指向的路径。若创建外部表，仅仅只是记录数据所在的位置（可以是 HDFS 目录或者本地目录），不对数据的位置做任何改变。在删除表的时候，内部表的元数据和数据都会被一起删除，而外部表仅仅只是改变元数据，不对数据进行任何操作。所以使用外部表相对更加安全，数据组织也更加灵活，方便共享数据。下面我们来看一下外部表的创建。假定我们有以下两个文件：

```
hadoop fs -ls /tmp/result/20160426
Found 2 items
03-rw-r--r--   3 bi supergroup      1240 2016-4-26 17:15
/tmp/result/20160426/part-00000
04-rw-r--r--   1 bi supergroup      1240 2016-4-26 17:58
/tmp/result/20160426/part-00001
```

下面创建外部表：

```
CREATE EXTERNAL TABLE IF NOT EXISTS test (userid string)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/tmp/result/20160426';
```

关键字 EXTERNAL 告诉 Hive 这是个外部表，后面的 LOCATION 告诉 Hive 数据位于哪个路径下。当删除表时，只会从 Hive 上删除表的元数据信息（删除链接信息），上述路径上的数据文件不会被删除。在 DESCRIBE EXTENDED test 语句的输出结果中会显示是内部表还是外部表。

Hive 有分区表的概念，这能提高数据库性能，主要是加快查询。我们来看一个例子：

```
CREATE TABLE employees (
empID   INT,
name    STRING,
salary  FLOAT,
skills     ARRAY<STRING>,
educations MAP<STRING, STRING>,
address    STRUCT<street:STRING, city:STRING>)
PARTITIONED BY (state STRING, city STRING);
```

在上述列子中，我们让 Hive 先按照 state（省）再按照 city（市）来对数据进行分区。那么，在这个表的目录下，将会出现反映分区的子目录，比如：

```
…/employees/state=ZJ/city=HZ
…/employees/state=ZJ/city=NP
…
```

每个城市的文件夹下包含着这个城市的员工信息。SHOW PARTITIONS employees 命令可以查看表中存在的所有分区。上述分区字段的使用方法类似普通的字段（即：分区字段可以作为 where 条件），比如：

```
SELECT *  FROM employees WHERE state='ZJ' AND city = 'HZ';
```

分区的主要好处是加快查询速度。比如，上述的查询中，我们只需要查询一个目录下的内容即可，这对于大数据集来说，能够极大的提高查询性能。我们还可以在使用 LOAD 命令加载数据时指定分区信息，外部表也可以有分区。

大多数的表属性可以通过 ALTER TABLE 语句来进行修改。比如：

```
ALTER TABLE employees CHANGE COLUMN empID employeeID INT AFTER name;
ALTER TABLE employees ADD COLUMNS ( age INT COMMENT 'employee age');
ALTER TABLE test add partition (hp_cal_dt='20160214') location
'/tmp/result/20160214';
```

在传统的 RDBMS 中，为了能够存储大量的数据，经常按月建立表，这样数据就可以分散在不同的月表中了。Hive 的分区表可以获得类似的好处，比如：

```
CREATE TABLE salesrecord (id int, ….) PARTIONED BY (int month);
ALTER TABLE salesrecord add PARTITION (month=201601);
ALTER TABLE salesrecord add PARTITION (month=201602);
ALTER TABLE salesrecord add PARTITION (month=201603);
……
SELECT … FROM salesrecord WHERE month>=201602 AND month<201603;
```

Hive 需要进行全表扫描来执行查询。通过创建多个分区就可以优化查询。需要注意的是，一个理想的分区设计不应该产生太多的分区和文件夹目录，并且每个目录下的文件应该足够得大。这是因为，HDFS 是用于存储海量的大文件，而不是海量的小文件。除了分区的功能，Hive 还提供了 buckets（分桶）的功能，能将数据集分解成更容易管理的若干部分。具体内容，可参考 Hive 网站。

Hive 没有主键或自增键。Hive 支持有限的索引功能，可以对一些字段建立索引来加速查询操作。一张表的索引数据存储在另外一张表中。

6.2.7 HiveQL DML

对于 SQL 而言，DML 就是增删改查的 SQL 语句。但是，Hive 没有行级别的数据插入、数据更新和数据删除操作，往表中装载数据的唯一途径就是使用 LOAD 命令来大量地数据装载。

比如：

```
LOAD DATA LOCAL INPATH '/home/work/test.txt' INTO TABLE MYTEST2;
```

如果使用了 LOCAL 关键字，那么，这个路径应该为本地文件系统路径。如果没有这个 LOCAL 关键字，那么这个路径就是分布式文件系统中的路径。

Hive 支持 INSERT..SELECT 语句，通过查询语句向表中插入数据。比如：

```
INSERT INTO TABLE employees SELECT * FROM old_employees …
```

上述的 INTO 可替换为 OVERWRITE，这样 Hive 就会覆盖之前已经存在的内容。INSERT..SELECT 语句也可以放在 CREATE TABLE 语句中，从而在一个语句中完成创建表并将查询结果插入这个表中，比如：

```
CREATE TABLE zj_employees AS
 SELECT empID,name,salary FROM employees WHERE …..
```

Hive 导出数据的方式有多种。比如：

```
bin/hive --database 'hrsystem' -e 'select * from employees' >>
/home/sam/exportDir/emp.tsv
hive  --database  'default'  -e  'select * from employees' >> /emp2.tsv
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/'  SELECT …
```

在 SELECT 语句中，常见的数学函数（如：abs()）、聚合函数（如：count()、sum()、avg()等）、字符串函数（如：concat、locate、substr 等）都可以使用。如果想要限制返回的行数，则可以使用 LIMIT 语句。SELECT 还支持嵌套 SELECT 语句和 CASE WHEN 句式。WHERE 子句支持常规的操作符（如：=、>、LIKE、IS NULL 等），也支持 GROUP BY、HAVING、JOIN、ORDER BY 等。

Hive 提供了 EXPLAIN 功能，解释 Hive 如何将查询转化成 MapReduce 任务的。比如：

```
EXPLAIN SELECT avg(salary) FROM employees;
```

在执行上述语句时，Hive 会打印出语法树，表明如何将查询解析成 token 和 literal 的。Hive 也会打印出多个 stage（一个 Hive 任务包含一个或多个 stage），不同的 stage 之间可能存在着依赖关系，一个 stage 可以是一个 MapReduce 任务或是其他的操作。EXPLAIN 返回的结果中可能还包含了 reduce 操作树（Reduce Operator Tree）。当执行具有 reduce 过程的 Hive 查询（如：带有 GROUP BY 子句的语句）时，CLI 控制台会打印出调优后的 reducer 个数。Hive 是按照输入的数据量大小来确定 reducer 个数的。你可以使用“dfs –count”命令来计算输入量的大小，这个命令同 Linux 中的 “du –s”命令类似，可以计算出指定目录下所有数据的总大小。Hive 有一个参数 hive.exec.reducers.bytes.per.reducer，其设置了一个 reducer 处理的数据量，默认为 1GB。而 hive.exec.reducers.max 则指定了最大的 reducer 个数。

6.2.8　Hive 编程

我们使用 JDBC 开发 Hive 程序，这和传统的 JDBC 开发没有太大的区别。要注意的是，在使用 JDBC 开发 Hive 时，首先需要开启 Hive 的远程服务接口。使用下面命令进行开启:

```
hive -service hiveserver &
```

HiveServer 也叫 HiveThrift，它允许通过指定端口访问 Hive。Thrift 是一个软件框架，支持 Java、C++等编程语言，通过编程的方式远程访问 Hive。下面是 Hive 的一个示例程序：

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import org.apache.log4j.Logger;

public class HiveJdbcClient {
    private static String driverName =
"org.apache.hadoop.hive.jdbc.HiveDriver";
    private static String url =
"jdbc:hive://192.168.11.157:10000/default";
    private static String user = "hive";
    private static String pwd = "mysql";
    private static String sql = "";
    private static ResultSet res;
    private static final Logger log =
Logger.getLogger(HiveJdbcClient.class);

    public static void main(String[] args) {
        try {
            Class.forName(driverName);
            Connection conn = DriverManager.getConnection(url,
user, pwd);
            Statement stmt = conn.createStatement();

            // 创建的表名
            String tableName = "testHiveDriverTable";
            //第一步:存在就先删除
            sql = "drop table " + tableName;
```

```
                    stmt.executeQuery(sql);

                    //第二步:创建表
                    sql = "create table " + tableName +
    " (key int, value string)  row format delimited fields terminated by
'\t'";
                    stmt.executeQuery(sql);

                    // 执行"show tables"操作
                    sql = "show tables '" + tableName + "'";
                    System.out.println("Running:" + sql);
                    res = stmt.executeQuery(sql);
                    System.out.println("执行"show tables"运行结果:");
                    if (res.next()) {
                           System.out.println(res.getString(1));
                    }

                    // 执行"describe table"操作
                    sql = "describe " + tableName;
                    System.out.println("Running:" + sql);
                    res = stmt.executeQuery(sql);
                    System.out.println("执行"describe table"运行结果:");
                    while (res.next()) {
                           System.out.println(res.getString(1)  +  "\t"  +
res.getString(2));
                    }

                    // 执行"load data into table"操作
                    String                    filepath                    =
"/home/hadoop/zhenghong/userinfo.txt";
                    sql = "load data local inpath '" + filepath + "' into
table " + tableName;
                    System.out.println("Running:" + sql);
                    res = stmt.executeQuery(sql);

                    // 执行"select * from"操作
                    sql = "select * from " + tableName;
                    System.out.println("Running:" + sql);
                    res = stmt.executeQuery(sql);
```

```
                    System.out.println("执行"select "运行结果:");
                    while (res.next()) {
                           System.out.println(res.getInt(1)    +    "\t"    +
res.getString(2));
                    }

                    // 执行"regular hive query"操作
                    sql = "select count(1) from " + tableName;
                    System.out.println("Running:" + sql);
                    res = stmt.executeQuery(sql);
                    System.out.println("执行"regular hive query"运行结果:");
                    while (res.next()) {
                           System.out.println(res.getString(1));
                    }

                    conn.close();
                    conn = null;
             } catch (ClassNotFoundException e) {
                    e.printStackTrace();
                    log.error(driverName + " not found!", e);
                    System.exit(1);
             } catch (SQLException e) {
                    e.printStackTrace();
                    log.error("Connection error!", e);
                    System.exit(1);
             }
        }
    }
```

从上看出，Hive 的编程同一般的 JDBC 编程没什么区别。

6.2.9 HBase 集成

HBase 与 Hive 都是架构在 HDFS 系统之上的 Hadoop 生态圈的组件，利用 HDFS 作为底层存储。我们把 HBase 看作分布式数据库，把 Hive 作为分布式数据仓库。Hive 的适用场景是非实时、面向批处理的工作，比如：海量数据的批量处理、统计查询和计算分析。HBase 的适用场景是实时处理工作，它作为 NoSQL 数据库，设计数据库的 Schema，处理高并发的实时快速查询和插入。我们可以将 HBase 和 Hive 结合起来使用。一种方式就是让 HBase 作实时处理，然后将 HBase 数据导入到 HDFS 文件上，最后通过 Hive 来做数据分析。这是因为 HBase 可能会对底层

多个文件合并，而从 HDFS 中访问是顺序 I/O，所以直接让 Hive 操纵 HDFS 上的海量数据分析会更快。还有，Hive 采用了类 SQL 的查询语言 HQL，它会自动转化成 MapReduce 程序，提高工程师的开发效率，同时系统易扩展和维护。

还有一种更加紧密的合作方式，在 Hive 上直接访问 HBase 表。我们只需要在 Hive 上创建一个指向 HBase 表的外部表即可。比如：

```
CREATE EXTERNAL TABLE hbase_stocks (key INT, name STRING, price FLOAT)
STORED BY 'org.apache.hadoop.hive.hbase.HBaseStorageHandler'
WITH SERDEPROPERTIES ("hbase.columns.mapping"="cf1:val")
TBLPROPERTIES("hbase.table.name"=stocks);
```

我们首先解释一下上述 CREATE 语句中出现的几个关键字：

- HBaseStorageHandler。Hadoop 有一个 InputFormat 抽象接口类，可以将来自不同数据源的数据格式化为 job 的输入格式；有一个 OutputFormat 抽象接口类，用于获得一个 job 输出，以写入到目标实体上。InputFormat 和 OutputFromat 可以是文件，也可以是 RDBMS 和 HBase 等。HiveStorageHandler 是 Hive 用于连接 HBase 的接口，里面有定制的 InputFromat、OutputFormat 和 SerDe。HBaseStoragehandler 实现了 HiveStorageHandler 接口。
- WITH SERDEPROPERTIES。Hive 支持不同格式的文件来存储数据。默认情况下是 TEXTFILE(文本文件)。SerDe 也是一种格式，是序列化/反序列化的简写。

Hive 与 HBase 的整合功能的实现是利用两者本身对外的 API 接口互相进行通信，相互通信主要是依靠 hive_hbase-handler.jar 工具类， 结构如图 6-20 所示。

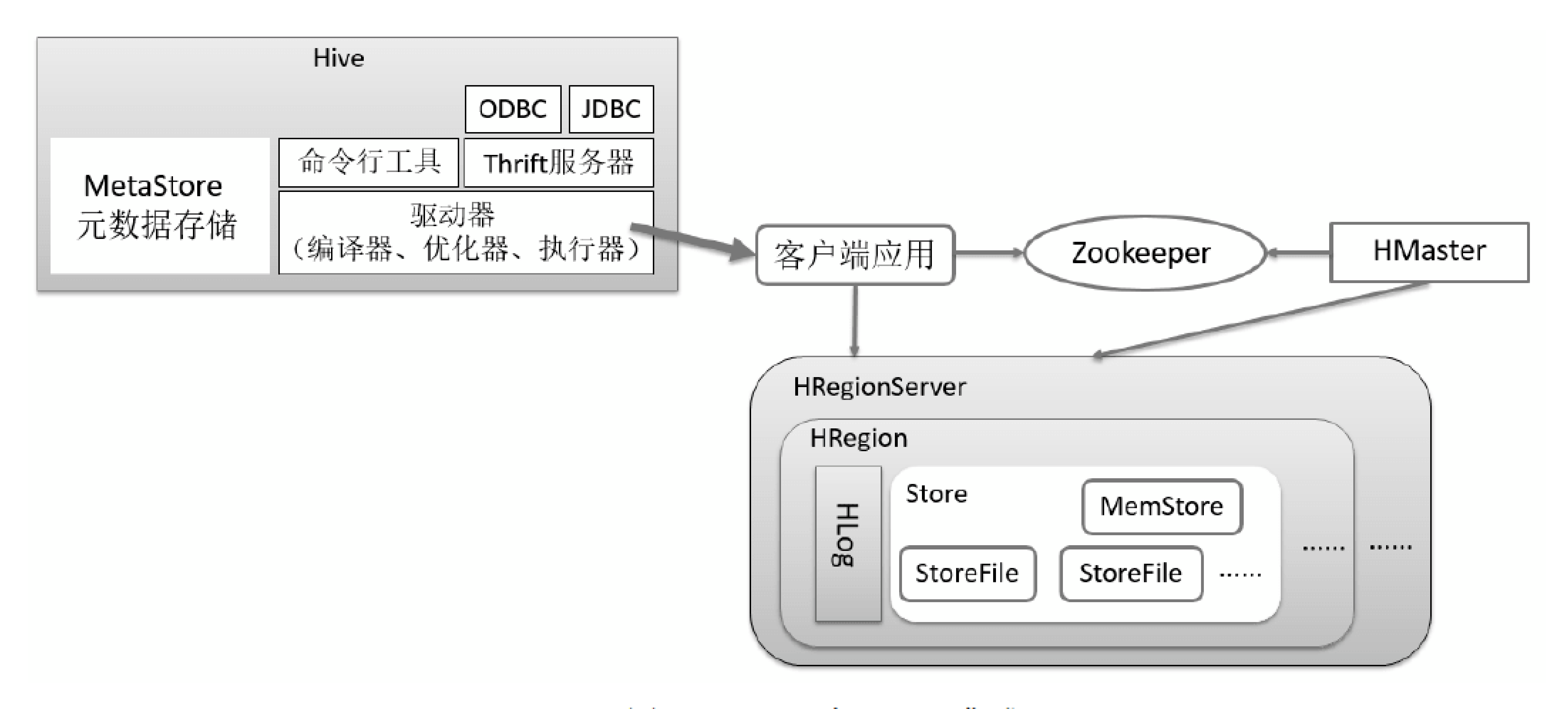


图 6-20　Hive 与 HBase 集成

在 Hive 的安装目录的 lib 子目录中，已经存在了 HBase 和 ZooKeeper 的相关 Jar，但是版本可能不是很一致，需要把 HBase 安装目录中的相关 Jar 拷贝到 Hive 的 lib 下面。重新编译“hive-

hbase-handler”这个 Jar 包。最后确保 Hive 运行正常：

```
hive> show tables;
OK
Time taken: 8.418 seconds
hive> create

create          create_union(
hive> create TABLE my(id INT,name string) ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
OK
Time taken: 1.872 seconds
hive> show tables;
OK
my
Time taken: 0.161 seconds, Fetched: 1 row(s)
hive>
```

6.2.10　XML 和 JSON 数据

Hive 包含了 XPath 相关的 UDF，可以从 XML 中提取数据。比如：

```
select xpath ('<a><b id="1"><c/></b><b id="2"><c/></b></a>' ,
'/descendant::c/ancestor::b/@id') from t1 limit 1 ;
```

具体函数如表 6-2 所示。

表 6-2　XPath 函数

UDF 名称	说明
xpath	返回 Hive 中的一组字符串数组
xpath_string	返回一个字符串
xpath_boolean	返回一个布尔值
xpath_short	返回一个短整型数值
xpath_int	返回一个整型数值
xpath_long	返回一个长整型数值
xpath_float	返回一个浮点数数值
xpath_double,xpath_number	返回一个双精度浮点数数值

JSON 是一种轻量级的数据格式，结构灵活，支持嵌套，非常易于阅读和编写，而且主流的编程语言都提供相应的框架或类库支持与 JSON 数据的交互。Hive 可以查询 JSON 格式的数据。我们可以创建一个外部表来指向 JSON 数据，比如：

```
CREATE EXTERNAL TABLE message(
msgID  INT,
crtTS   STRING,
text    STRING,
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.JsonSerde'
```

```
WITH SERDEPROPERTIES (
 "msgID"="$.id",
 "crtTS"="$.created_at",
 "text"="$.text"
)
LOCATION '/data/jsondata';
```

上例中的属性用于将 JSON 文档和表的列对应起来。一旦定义好之后，用户就可以执行查询，而不用关心查询是如何从 JSON 中获取数据的。

6.2.11 使用 Tez

Tez 是 Apache 最新开源的支持 DAG 作业的计算框架，它可以将多个有依赖的作业转换为一个作业，从而大幅提升 DAG 作业的性能。它直接源于 MapReduce 框架，核心思想是将 Map 和 Reduce 两个操作进一步拆分，即 Map 被拆分成 Input、Processor、Sort、Merge 和 Output，Reduce 被拆分成 Input、Shuffle、Sort、Merge、Processor 和 Output 等，这些分解后的各个操作可以任意灵活组合，产生新的操作，这些操作经过一些控制程序组装后，可形成一个大的 DAG 作业。总结起来，Tez 运行在 YARN 之上，适用于 DAG（有向图）应用（同 Impala、Dremel 和 Drill 一样，可用于替换 Hive/Pig 等）。Hive on Tez 是以 Tez 为计算框架的 hive 数据分析系统。

通过 Tez 我们可以构建性能更快、扩展性更好的应用程序。Hadoop 传统上是一个海量数据批处理平台。但是，很多场景需要近乎实时的查询处理性能。还有一些工作则不太适合 MapReduce，例如机器学习。Tez 的目的就是帮助 Hadoop 处理这些用例场景。Tez 项目的目标是支持高度定制化，满足各种用例的需要，让人们不必借助其他的外部方式就能完成自己的工作。如果 Hive 和 Pig 项目使用 Tez 而不是 MapReduce 作为其数据处理的骨干，那么将会显著提升它们的响应时间。Tez 构建在 YARN 之上，后者是 Hadoop 所使用的新资源管理框架。

为了在 Hive 上使用 Tez，你只需要设置如下参数：

```
set hive.execution.engine=tez
```

下面我们做两个简单的测试，一个是在 MapReduce 上执行统计计算，另一个是在 Tez 上，我们最终发现 Hive 在 Tez 上的性能有很大提升。

在 MapReduce 上测试：

```
hive> select count(*) from wyp4;
Query ID = hive_20141215111818_cbdf0e31-4a40-4153-97a3-0df11d2b0dd8
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
```

```
    In order to limit the maximum number of reducers:
      set hive.exec.reducers.max=<number>
    In order to set a constant number of reducers:
      set mapreduce.job.reduces=<number>
    Starting    Job    =    job_1418266580454_0004,    Tracking    URL    =
http://phicomm.hdp2:8088/proxy/application_1418266580454_0004/
    Kill  Command  =  /usr/hdp/2.2.0.0-2041/hadoop/bin/hadoop  job   -kill
job_1418266580454_0004
    Hadoop job information for Stage-1: number of mappers: 1; number of
reducers: 1
    2014-12-15 11:18:50,075 Stage-1 map = 0%,  reduce = 0%
    2014-12-15 11:18:56,466 Stage-1 map = 100%,  reduce = 0%, Cumulative
CPU 1.48 sec
    2014-12-15 11:19:02,758 Stage-1 map = 100%,  reduce = 100%, Cumulative
CPU 2.95 sec
    MapReduce Total cumulative CPU time: 2 seconds 950 msec
    Ended Job = job_1418266580454_0004
    MapReduce Jobs Launched:
    Stage-Stage-1: Map: 1  Reduce: 1  Cumulative CPU: 2.95 sec  HDFS Read:
278 HDFS Write: 2 SUCCESS
    Total MapReduce CPU Time Spent: 2 seconds 950 msec
    OK
    3
    Time taken: 20.87 seconds, Fetched: 1 row(s)
```

在 TEZ 上测试：

```
    hive> set hive.execution.engine=tez;
    hive> select count(*) from wyp4;
    Query ID = hive_20141215112121_9a7252c7-77ff-4ed5-861f-08929d8124bf
    Total jobs = 1
    Launching Job 1 out of 1

    Status:   Running   (Executing   on   YARN   cluster   with   App   id
application_1418266580454_0005)

    ----------------------------------------------------------------------
-------------------------------------------------------
      VERTICES  STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED  KILLED
    ----------------------------------------------------------------------
```

```
------------------------------------------------------------
   Map 1 ..........  SUCCEEDED    1       1       0       0      0      0
   Reducer 2 ....  SUCCEEDED    1       1       0       0      0      0
   ----------------------------------------------------------------------------
------------------------------------------------------------
   VERTICES:  02/02   [==========================>>]  100%   ELAPSED  TIME:
4.89 s
   ----------------------------------------------------------------------------
-------------------------------------------------------------
   OK
   3
   Time taken: 10.834 seconds, Fetched: 1 row(s)
```

6.3 Pig

Pig 最初由 Yahoo 于 2006 年开发，当时是为了能够在大数据集上创建和执行 map-reduce 任务。 在 2007 年，Pig 成为 Apache Software Foundation 的一员。它在 MapReduce 框架上实现了一套 shell 脚本，在 Pig 中称之为 Pig Latin。Pig 将脚本转换为 MapReduce 任务，最后在 Hadoop 上执行。

假定某用户的输入数据来自多个源，而用户需要进行一组复杂的转换来生成一个或多个输出数据集。如果使用 Hive，则用户可能需要创建临时表和使用复杂的查询来完成。Pig 是一种过程语言，类似于存储过程，是对于输入的一步步操作，其中每一步都是对数据的一个简单的变换。在 Pig 脚本中，我们可以对加载进来的数据进行排序、过滤、求和、分组（group by）、关联，Pig 也可以调用用户自定义函数对数据集进行操作。Pig 适合即时性的数据处理需求，可以通过 Pig 很快写一个脚本开始运行处理，而不需要创建表等相关的事先准备工作。`

Pig 的使用主要是数据的 ETL。让我们看一个简单的 Pig 示例。假定我们要在大型数据集中搜索满足某个给定搜索条件的记录（类似 UNIX 中的 grep 命令）。Pig 的代码如下：

```
messages = LOAD 'messages';
warns = FILTER messages BY $0 MATCHES '.*WARN+.*';
STORE warns INTO 'warnings';
```

在上面代码中，第一行是装载数据集，第二行是用一个正则表达式来筛选该数据集，即查找字符序列 WARN。最后是将筛选结果存储在一个名为 warnings 的新文件中。这个简单的脚本实现了一个简单的流，但是，如果直接在传统的 MapReduce 模型中实现它，则需要增加大量的代码。

6.3.1　Pig 语法

Pig Latin 语言和传统的数据库语言很相似，但是 Pig Latin 更侧重于数据查询，而不是数据的修改和删除等操作。Pig 语句通常按照如下的流程来编写：

- 通过 LOAD 语句从文件系统读取数据；
- 通过一系列“转换”语句对数据进行处理；
- 通过一条 STORE 语句把处理结果输出到文件系统中，或者使用 DUMP 语句把处理结果输出到屏幕上。

Pig 的数据类型分为两类，分别为简单类型和复杂类型。简单类型包括：int、long、float、double、chararray、bytearray、boolean、datetime、biginteger、bigdecimal。复杂类型包括：tuple、bag、map。chararray 相当于字符串 String；bytearray 相当于字节数组。tuple 是一个有序的字段的集合，可以理解为元组，例如(100,‘Android’,50)。bag 是 tuple 的集合，例如{(100,‘Android’, 50), (101,‘iOS’,60)}。map 是“键-值”对的集合，例如[name#zhenghong, age#38, healthy index#195]。

下面我们给出 Pig 的一些常见用法。为了方便读者的理解，我们同时给出 MySQL 的语法。

1. 从文件导入数据

（1）MySQL （MySQL 需要先创建表）

```
CREATE TABLE TMP_TABLE(USER VARCHAR(32),AGE INT,IS_MALE BOOLEAN);
CREATE TABLE TMP_TABLE_2(AGE INT,OPTIONS VARCHAR(50)); -- 用于 Join
LOAD DATA LOCAL INFILE '/tmp/data_file_1' INTO TABLE TMP_TABLE ;
LOAD DATA LOCAL INFILE '/tmp/data_file_2' INTO TABLE TMP_TABLE_2;
```

（2）Pig

```
tmp_table = LOAD '/tmp/data_file_1' USING PigStorage('\t') AS
(user:chararray, age:int,is_male:int);
tmp_table_2= LOAD '/tmp/data_file_2' USING PigStorage('\t') AS
(age:int, options:chararray);
```

2. 查询整张表

（1）MySQL

```
SELECT * FROM TMP_TABLE;
```

（2）Pig

```
DUMP tmp_table;
```

DUMP 命令用于在屏幕上显示数据。

3. 查询前 50 行

（1）Mysql

```
SELECT * FROM TMP_TABLE LIMIT 50;
```

（2）Pig

```
tmp_table_limit = LIMIT tmp_table 50;
DUMP tmp_table_limit;
```

4. 查询某些列

（1）Mysql

```
SELECT USER FROM TMP_TABLE;
```

（2）Pig

```
tmp_table_user = FOREACH tmp_table GENERATE user;
DUMP tmp_table_user;
```

5. 排序

（1）Mysql

```
    SELECT * FROM TMP_TABLE ORDER BY AGE;
```

（2）Pig

```
tmp_table_order = ORDER tmp_table BY age ASC;
DUMP tmp_table_order;
```

6. 条件查询

（1）Mysql

```
     SELECT * FROM TMP_TABLE WHERE AGE>18;
```

（2）Pig

```
tmp_table_where = FILTER tmp_table by age > 18;
DUMP tmp_table_where;
```

7. 内连接（Inner Join）

（1）Mysql

```
SELECT * FROM TMP_TABLE A JOIN TMP_TABLE_2 B ON A.AGE=B.AGE;
```

（2）Pig

```
tmp_table_inner_join = JOIN tmp_table BY age,tmp_table_2 BY age;
DUMP tmp_table_inner_join;
```

8. 左外连接（Left OUTER Join）

（1）Mysql

```
SELECT * FROM TMP_TABLE A LEFT JOIN TMP_TABLE_2 B ON A.AGE=B.AGE;
```

（2）Pig

```
tmp_table_left_join = JOIN tmp_table BY age LEFT OUTER,tmp_table_2 BY
age;
DUMP tmp_table_left_join;
```

9. 右外连接（Right OUTER Join）

（1）Mysql

```
SELECT * FROM TMP_TABLE A RIGHT JOIN TMP_TABLE_2 B ON A.AGE=B.AGE;
```

（2）Pig

```
tmp_table_right_join = JOIN tmp_table BY age RIGHT OUTER,tmp_table_2
BY age;
DUMP tmp_table_right_join;
```

10. 分组（GROUP BY）

（1）Mysql

```
SELECT * FROM TMP_TABLE GROUP BY IS_MALE;
```

（2）Pig

```
tmp_table_group = GROUP tmp_table BY is_male;
DUMP tmp_table_group;
```

GROUP BY 是根据某个或某些字段进行分组，只根据一个字段进行分组，比较简单。如果想要根据两个字段分组，则可以将两个字段构造成一个 tuple，然后进行分组。

11. 统计（COUNT）

（1）Mysql

```
SELECT IS_MALE,COUNT(*) FROM TMP_TABLE GROUP BY IS_MALE;
```

（2）Pig

```
tmp_table_group_count = GROUP tmp_table BY is_male;
tmp_table_group_count = FOREACH tmp_table_group_count GENERATE
group,COUNT($1);
DUMP tmp_table_group_count;
```

FOREACH 操作可以针对一个数据集进行迭代处理操作，生成一个新的数据集。

12. 查询去重（DISTINCT）

（1）MYSQL

```
SELECT DISTINCT IS_MALE FROM TMP_TABLE;
```

（2）Pig

```
tmp_table_distinct = FOREACH tmp_table GENERATE is_male;
tmp_table_distinct = DISTINCT tmp_table_distinct;
DUMP  tmp_table_distinct;
```

在进行数据过滤时，建议尽早使用 foreach generate 将多余的数据过滤掉，减少数据交换。

6.3.2 Pig 和 Hive 的使用场景比较

很多的项目都想用 Hadoop 作为数据存储，而以 SQL 构建前端查询。为了简化 Hadoop 的使用，出现了类似于 SQL 的 Pig 和 Hive。而用户在进行大数据处理的时候，使用这些工具可以避免复杂的 Java 编码，但在使用之前很重要的一点是了解工具之间的区别，以便在不同的用例中使用最优化的工具。选对平台和语言对于数据的提取、处理和分析都起着至关重要的作用。

SQL 程序员们需要这样一种编程语言：既利于 SQL 程序员们学习同时又能够轻松应对大型数据集。Pig 很好地解决了上面提到的问题，同时也提供了较好的扩展性和性能优化。Apache Pig 对 Multi-query 的支持减少了数据检索循环的次数。Pig 支持 map、tuple 和 bag 这样的复合数据类型以及常见的数据操作如筛选、排序和联合查询。这些优势让 Pig 在全球范围内都得到了广泛的应用。Pig 简便的特点也是雅虎和 Twitter 使用它的原因之一。

尽管 Pig 性能强劲，如果要使用它，开发人员则必须掌握 SQL 之外的新知识，而 Hive 则与 SQL 非常相像。尽管 Hive 查询语言 HQL 的命令有所局限，它还是取得了一定的成功。Hive 为 MapReduce 提供了优秀的开源实现，它在分布式数据处理的同时避免了 SQL 对于数据存储的局限。

下面我们就把 Pig、Hive 和 SQL 两两进行对比以便了解它们各自所适用的情况。

- Pig vs SQL

与单纯的 SQL 相比，Pig Latin 在声明式执行计划、ETL 流程和管道的修改上则有着优势。整体上来看，SQL 是一门声明式语言，而 PigLatin 属于过程式语言。在 SQL 中我们指定需要完成的任务，而在 Pig 中我们则指定任务完成的方式。Pig 脚本其实都是转换成 MapReduce 任务来执行的，不过 Pig 脚本会比对应的 MapReduce 任务简短很多，所以开发的速度要快很多。

- Hive vs SQL

SQL 是一门通用的数据库语言，大量的事务和分析语句都是由 SQL 完成的。Hive 则是以数据分析为目标所设计的，这意味着虽然 Hive 缺乏更新和删除这样的功能，但读取和处理大量数据的速度会比 SQL 快得多。所以 Hive SQL 看起来像 SQL，但在更新和删除等功能上两者还是有很大区别的。虽然有所不同，但如果你有 SQL 背景的话学习起 Hive 还是很容易的。不过要注意两者在构造和语法上的区别，否则容易混淆。

下面我们来分析以下三种语言最适用的情况。

1. 什么时候用 Apache Pig

当你需要处理非格式化的分布式数据集时，如果想充分利用自己的 SQL 基础，可以选择 Pig。使用 Pig 你无须自己构建 MapReduce 任务，有 SQL 背景的话学习起来比较简单，开发速度也很快。

2. 什么时候用 Apache Hive

有时我们需要收集一段时间的数据来进行分析，而 Hive 就是分析历史数据绝佳的工具。要注意的是，数据必须有一定的结构才能充分发挥 Hive 的功能。用 Hive 来进行实时分析可能就不是太理想了，因为它不能达到实时分析的速度要求（实时分析可以用 HBase，Facebook 用的就是 HBase）。

3. 什么时候用 SQL

SQL 是这三者中最传统的数据分析手段。随着用户需求的改变，SQL 本身也在进行着更新，即便到了今天也不能说 SQL 过时。用 SQL 来进行快速的复杂处理和分析还是显得有点欠缺。如果所进行的分析比较简单的话，SQL 仍然是一个非常好的工具。大部分开发人员都对 SQL 有所了解，所以使用 SQL 的话，开发人员从项目开始的第一天就能有所产出。SQL 提供的扩展和优化功能也让我们能够根据需求进行定制。

不同的数据没有一个所有情况都适用的查询工具，根据自己的需求来选择不同工具才是正确的方法。

6.4 ElasticSearch（全文搜索引擎）

Elasticsearch（简称为 ES）是一个实时的分布式搜索和分析引擎。它可以快速处理大规模数据，用于全文搜索，结构化搜索以及分析。Elasticsearch 是一个建立在全文搜索引擎 Apache Lucene 基础上的搜索引擎，而 Lucene 是当今最先进、最高效的全功能开源搜索引擎框架。因为 Lucene 只是一个框架，要充分利用它的功能，需要使用 Java 在程序中集成 Lucene。而 Elasticsearch 使用 Lucene 作为内部引擎，但是在使用它做全文搜索时，只需要使用统一开发好的 API 即可，而不需要了解其背后复杂的 Lucene 运行原理。

Elasticsearch 是用 Java 开发的，它作为 Apache 许可条款下的开放源码发布，是一个完全免费的搜索产品，也是当前流行的企业级搜索引擎，能够实现实时搜索，它稳定、可靠、快速、安装使用方便。它可运行在 Windows 和 Linux 之上。Elasticsearch 包括了全文搜索功能，而且还包括了以下功能：

- 分布式实时文件存储，并将每一个字段都编入索引，使其可以被搜索。
- 实时分析的分布式搜索引擎。
- 能够从一台扩展到上百台服务器，处理 PB 级别的结构化或非结构化数据。
- 通过客户端或者程序语言与 ES 的 RESTful API 进行交流。

6.4.1 全文索引的基础知识

全文索引是一种数据结构，它允许对存储在文件中的单词进行快速随机访问。当需要从大量文件中快速检索文本目标时，首先必须将文件内容转换成能够进行快速搜索的格式，以建立针对文件的索引数据结构，此即为索引过程。它通常由逻辑上互不相关的下面几个步骤组成。

1. 获取原始内容

我们可以通过网络爬虫等应用来搜集需要索引的内容。例如著名的开源爬虫程序有 Solr、Nutch 等等。

2. 建立文档

获取的原始内容需要转换成文档才能提供给搜索引擎使用。一般来说，一个网页、一个 PDF 文档、一封邮件或一条日志信息都可以作为一个文档。

3. 文档分析

搜索引擎不能直接对文本进行索引，确切地说，必须首先将文本分割成一系列被称为词汇单元（token）的独立原子元素，此过程即为文档分析。每个 token 大致能与自然语言中的“单词”对应起来，文档分析就是用于确定文档中的文本域如何分割成 token 序列。

4. 文档索引

在索引步骤中，文档将被加入到索引列表。

我们以 ES 为例，来看一下上述的第 3 步和第 4 步。如图 6-21 所示，ES 分析文本并将其构建成为倒排索引（inverted index），倒排索引由各文档中出现的单词列表组成，列表中的各单词不能重复且需要指向其所在的各文档。因此，为了创建倒排索引，需要先将各文档中域的值切分为独立的单词（也称为 term 或 token），而后将之创建为一个无重复的有序单词列表。这个过程称之为“分词（tokenization）”。

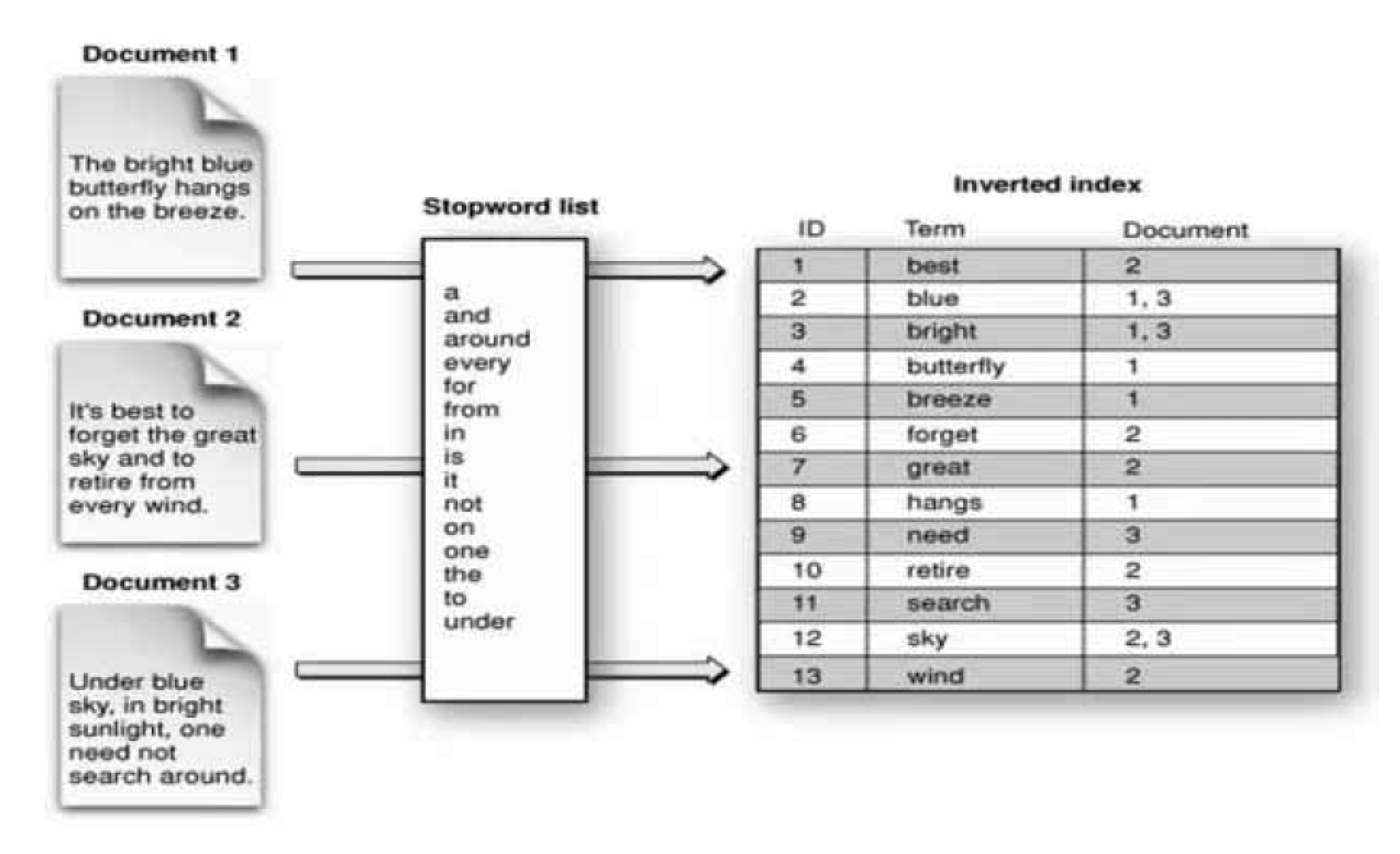


图 6-21　全文索引示例

为了实现全文索引，倒排索引中的数据还需进行“正规化（normalization）”为标准格式，才能评估其与用户搜索请求字符串的相似度。例如，对于英文全文检索，将所有大写字符转换为小写，将复数统一单数，将同义词统一进行索引等。另外，执行查询之前，还需要将查询字符串按照索引过程的同种格式进行“正规化（normalization）”。这里的“分词”及“正规化”操作也称为“分析（analysis）”。Analysis 过程由两个步骤组成：首先将文本切分为 terms（词项）以适合构建倒排索引，其次将各 terms 正规化为标准形式以提升其“可搜索度”。这两个步骤由分析器（analyzers）完成。一个分析器通常需要由三个组件构成：字符过滤器（Character filters）、分词器（Tokenizer）和分词过滤器（Token filters）组成。

- 字符过滤器：在文本被切割之前进行清理操作，例如移除 HTML 标签，将&替换为字符等；
- 分词器：将文本切分为独立的词项；简单的分词器通常是根据空白及标点符号进行切分；
- 分词过滤器：转换字符（如将大写转为小写）、移除词项（如移除 a、an、of 及 the 等）或者添加词项（例如，添加同义词）。

Elasticsearch 内置了许多字符过滤器、分词器和分词过滤器，用户可按需将它们组合成自定义的分析器。固然，创建倒排索引时需要用到分析器，但传递搜索字符串时也可能需要分析器，甚至还要用到与索引创建时相同的分析器才能保证单词匹配的精确度。

全文检索就是从索引中查找词汇，从而找到包含该词汇的文档的过程。除了快速搜索大量文本和搜索速度之外，搜索过程还涉及了许多其他问题，例如单项查询、多项查询、短语查询、通配符查询、结果 ranking 和排序，以及友好的查询输入方式等。

6.4.2 安装和配置 ES

ES 的上手比较简单，它是一个零配置系统。它附带了一些非常合理的默认值，安装好了就可以立即使用。随着深入学习和使用，还可以利用 Elasticsearch 更多高级的功能，整个引擎可以很灵活地进行配置，可以根据自身需求来定制属于自己的 Elasticsearch。

很多用户使用 ES 来完成全文搜索功能。比如：维基百科使用 Elasticsearch 来进行全文搜索并高亮显示关键词，以及提供 search-as-you-type、did-you-mean 等搜索建议功能。StackOverflow 将全文搜索与地理位置和相关信息进行结合，以提供 more-like-this 相关问题的展现。2013 年初，GitHub 抛弃了 Solr，采取 ElasticSearch 来做 PB 级的搜索。据称，GitHub 使用 ElasticSearch 搜索 20TB 的数据，包括 13 亿文件和 1300 亿行代码。高盛（Goldman Sachs）每天使用它来处理 5TB 数据的索引，还有很多投资银行使用它来分析股票市场的变动。SoundCloud 使用 ElasticSearch 为 1.8 亿用户提供即时而精准的音乐搜索服务。而百度使用 ElasticSearch 作为文本数据分析，采集百度所有服务器上的各类指标数据及用户自定义数据，通过对各种数据进行多维分析展示，辅助定位分析实例异常或业务层面异常。目前覆盖百度内部 20 多个业务线，单集群最大 100 台机器，200 个 ES 节点，每天导入 30TB+数据。

近年 ElasticSearch 发展迅猛，已经超越了其最初的纯搜索引擎的角色，现在已经增加了数据聚合分析（aggregation）和可视化的特性。如果你有数百万的文档需要通过关键词进行定位时，ElasticSearch 肯定是最佳选择。如果你的文档是 JSON 的，你也可以把 ElasticSearch 当作一种“NoSQL 数据库”，应用 ElasticSearch 数据聚合分析的特性，针对数据进行多维度的分析。

ElasticSearch 可以以单点或者集群方式运行，以一个整体对外提供 search 服务的所有节点组成 cluster，组成这个 cluster 的各个节点叫做 node。ElasticSearch 使用 index 存储索引数据，类似于一个数据库。ElasticSearch 使用索引分片（Shards），这是 ES 提供分布式搜索的基础，它将一个完整的 index 分成若干部分存储在相同或不同的节点上，这些组成 index 的部分就叫做 shard。

ElasticSearch 可以设置多个索引的副本（Replicas），副本的作用是提高系统的容错性，当某个节点某个分片损坏或丢失时可以从副本中恢复。另外，它可以提高 ElasticSearch 的查询效率，ES 会自动对搜索请求进行负载均衡。ElasticSearch 还提供数据恢复（Recovery，也叫数据重新分布），它在有节点加入或退出时会根据机器的负载对索引分片进行重新分配，挂掉的节点重新启动时也会进行数据恢复。

Gateway 是 ElasticSearch 索引快照的存储方式，使得备份更加简单。ES 默认是先把索引存

放到内存中，当内存满了的时候，再持久化到本地硬盘。gateway 对索引快照进行存储，当这个 ElasticSearch 集群关闭再重新启动时就会从 gateway 中读取索引备份数据。还有，ElasticSearch 是一个基于点对点的系统，它先通过广播寻找存在的节点，再通过多播协议来进行节点之间的通信，同时也支持点对点的交互。在 ElasticSearch 内部节点之间是使用 tcp 协议进行交互，同时它支持 http 协议（json 格式）。ElasticSearch 各节点组成对等的网络结构，某些节点出现故障时会自动分配其他节点代替其进行工作。

ES 由 Java 语言实现，运行环境依赖 Java，ES 推荐 Java 8 update 20 或更高，或 Java 7 update 55 或更高。ES 的最新版本是 2.3，可以去官网 https://www.elastic.co/products/elasticsearch/ 下载。下载后解压文件，它在指定路径生成 elasticsearch 相关目录。然后配置 ES，修改 ES_HOME/config/elasticsearch.yml 文件，配置格式是 YAML，比如：

```
#集群名称
cluster.name: elasticsearch
#节点名称
node.name: "node1"
#索引分片数
index.number_of_shards: 5
#索引副本数
index.number_of_replicas: 1
#数据目录存放位置
path.data: /data/elasticsearch/data
#日志数据存放位置
path.logs: /data/elasticsearch/log
```

进入 ES 安装目录，执行命令：bin/elasticsearch 就可以启动 ES，然后在浏览器输入 http://ip 地址:9200/，查看页面信息是否正常启动。status=200 表示正常启动了。接下来就是创建索引和对数据批量索引。之后就可以检索数据。当数据更新或删除时，可以更新或删除相应的索引。ElasticSearch 客户端支持多种语言，如 PHP、Java、Python、Perl 等。

检索速度快慢与索引质量有很大的关系。而索引质量的好坏主要与以下几方面有关。

1. 分片数

分片数是与检索速度非常相关的指标，如果分片数过少或过多都会导致检索比较慢。分片数过多会导致检索时打开比较多的文件，导致多台服务器之间通信。而分片数过少会导致单个分片索引过大，所以检索速度慢。在确定分片数之前，需要进行单服务单索引单分片的测试，从而确定单个分片的大小。

2. 副本数

副本数与索引的稳定性有比较大的关系，如果 Node 非正常挂了，这会导致分片丢失，为了保证这些数据的完整性，可以通过副本来解决这个问题。

3. 分词

分词对于索引有一定的影响。有人认为词库越大，分词效果越好，索引质量越好，其实不然。分词有很多算法，大部分基于词表进行分词。也就是说词表的大小决定索引大小。所以分词与索引膨胀率有直接关系。词表不应很多，采用对文档相关特征性较强的那种即可。在保证查全查准的情况下，词表数量越小，索引的大小可以减少越多。索引大小减少了，那么检索速度也就相应地提高了。

4. 内存优化

首先，ES 作为一个 Java 应用，就脱离不开 JVM 和 GC。在使用 ES 的过程中，要防止诸如 heap 不够用、内存溢出这样的问题，要知道哪些设置和操作容易造成以上问题，有针对性地予以规避。比如：ES 的底层引擎是基于 Lucene 的，而 Lucene 的倒排索引（Inverted Index）是先在内存里生成，然后定期以段文件（segment file）的形式存到磁盘的。API 层面的文档更新和删除实际上是增量写入的一种特殊文档，会保存在新的段里。不变的段文件易于被操作系统缓存。我们建议 heap size 不要超过系统可用内存的一半。其他的内存空间让 OS 来缓存段文件。而 JVM 参数并不需要做特别调整，可将 xms 和 xmx 设置成和 heap 一样大小，避免动态分配 heap size。可以使用 API 查看一个索引所有 segment 的内存占用情况，也可查看一个 node 上所有 segment 占用的内存总和，通过以下几个方法来减少 data node 上的 segment memory 占用：

- 删除不用的索引。
- 关闭索引（文件仍然存在于磁盘，只是释放掉内存）。需要的时候可以重新打开。
- 定期对不再更新的索引合并优化。这是对 segment file 强制合并，可以节省大量的 segment memory。

在开发与维护过程中，我们总结出以下优化建议：

- ES 性能体现在分布式计算中，一个节点是不足以测试出其性能，一个生产系统至少在三个节点以上。
- 倒排词典的索引需要常驻内存，无法 GC，需要监控 data node 上 segment memory 增长趋势。
- 根据机器数、磁盘数、索引大小等硬件环境，根据测试结果，设置最优的分片数和备份数，定期删除不用的索引，做好冷数据的迁移。
- 必须结合实际应用场景，对集群使用情况做持续的监控。

6.4.3 ES API

ES 提供了易用且功能强大的 RESTful API，用来与集群进行交互，这些 API 大体可分为如下四类：

- 检查集群、节点、索引等健康与否，以及获取其相关状态与统计信息;
- 管理集群、节点、索引数据及元数据;

- 执行 CRUD 操作及搜索操作；
- 执行高级搜索操作，例如 paging、filtering、scripting、faceting、aggregations 及其他操作。

ES 的 RESTful API 通过 TCP 协议的 9200 端口提供，客户端工具与此接口进行交互。一个流行的工具为 curl。curl 与 ElasticSearch 交互的通用请求格式如下所示：

```
curl -X<VERB> '<PROTOCOL>://<HOST>/<PORT>?<QUERY_STRING>' -d '<BODY>'
```

其中：

- VERB：HTTP 协议的请求方法，常用的有 GET、POST、PUT、HEAD 以及 DELETE;
- PROTOCOL：协议类型，http 或 https;
- HOST：ES 集群中的任一主机的主机名；
- PORT：ES 服务监听的端口，默认为 9200;
- QUERY_STRING：查询参数，例如?pretty 表示使用易读的 JSON 格式输出；
- BODY：JSON 格式的请求主体；

与 ElasticSearch 集群交互时，其输出数据均为 JSON 格式，多数情况下，此格式的易读性较差。cat API 会在交互时以类似于 Linux 上 cat 命令的格式对结果进行逐行输出，因此有着较 JSON 好些的可读性。调用 cat API 仅需要向“_cat”资源发起 GET 请求即可。

ES 中的数据查询（Query）API 是 ElasticSearch 的 API 中较大的一部分，可完成诸多类型查询操作，例如 simple term query、phrase、range 、boolean 、fuzzy、span 、wildcard 、spatial 等简单类型查询、组合简单查询类型为复杂类型查询，以及文档过滤等。另外，查询执行过程通常要分成两个阶段：分散阶段及合并阶段。分散阶段是向所查询的索引中的所有 shard 发起执行查询的过程，合并阶段是将各 shard 返回的结果合并、排序并响应给客户端的过程。向 ElasticSearch 发起查询操作有两种方式：一是通过 RESTful request API 传递查询参数，也称“query-string”；另一个是通过发送 REST request body，也称作 JSON 格式。通过发送 request body 的方式进行查询，可以通过 JSON 定义查询体，编写更具表现形式的查询请求。访问 ElasticSearch 的 search API 需要通过“_search”端点进行。例如，向 students 索引发起一个空查询：

```
curl -XGET 'localhost:9200/students/_search?pretty'
```

上面的查询命令也可改写为带 request body 的格式，其等同效果的命令如下：

```
curl -XGET 'localhost:9200/students/_search?pretty' -d '
{
  "query": { "match_all": { } }
}'
```

上述命令所示的查询语句是 ElasticSearch 提供的 JSON 风格的域类型查询语言，也即所谓的 Query DSL。上面的命令中，“query”参数给出了查询定义，match_all 给出了查询类型，它表示返回给定索引的所有文档。除了 query 参数之外，还可以指定其他参数来控制搜索结果，例如“size”参数可定义返回的文档数量（默认为 10），而“from”参数可指定结果集中要显示出的文档的起始偏移量（默认为 0），“sort”参数可指明排序规则等。

ElasticSearch 的大多数 search API（除了 Explain API）都支持多索引（mutli-index）和多类型（multi-type）。如果不限制查询时使用的索引和类型，查询请求将发给集群中的所有文档。ElasticSearch 会把查询请求并行发给所有 shard 的主 shard 或某一副本 shard。不过，如果是想向某一或某些索引的某一或某些类型发起查询请求，可通过指定查询的 URL 进行。比如：

```
/_search：搜索所有索引的所有类型；
/students/_search：搜索 students 索引的所有类型；
/students,tutors/_search：搜索 students 和 tutors 索引的所有类型；
/s*,t*/_search：搜索名称以 s 和 t 开头的所有索引的所有类型；
/students/class1/_search：搜索 students 索引的 class1类型；
/_all/class1,class2/_search：搜索所有索引的 class1和 class2类型；
```

Elasticsearch 支持许多的 query 和 filter。Filter DSL 中常见的有 term Filter、terms Filter、range Filter、exists and missing Filters 和 bool Filter。而 Query DSL 中常见的有 match_all、match、multi_match 及 bool Query。

第 7 章 大数据采集和导入

任何完整的大数据平台，一般包括以下的几个过程：数据采集、数据存储、数据管理、数据处理、数据展现（可视化、报表和监控）。数据是分散在不同的系统中的，在让数据产生价值之前，必须对数据进行采集、清洗、处理。大数据的数量和维度越来越多，我们必须采用大数据技术获取所需信息。计算机网络和信息设备的快速发展，产生的海量数据存在于各类服务器、媒介、机构，需要采取不同的办法去寻找、加工数据才可以获得所需信息。数据采集是所有数据系统必不可少的，随着大数据越来越被重视，数据采集的挑战也变得尤为突出。这其中包括：

- 数据源多种多样。
- 数据量大，变化快。
- 如何保证数据采集的可靠性，高性能。
- 如何避免重复数据。
- 如何保证数据的质量。

10 年前，网站日志是给开发人员和网站管理人员解决网站的问题。时至今日，网站日志数据可能包含了大量的业务和客户相关的很有价值信息，成为大数据分析的源数据。大数据采集首先是从网站日志收集开始的，之后进入了广阔的领域。本章以日志采集作为实例讲解大数据采集。正如我们在第 4 章中所阐述的，将数据存储到 HDFS 并不是难事，只需要使用一条“hadoop fs”命令即可。但是，这些网站一直在产生大量的日志（一般为流式数据），那么，使用上述命令批量加载到 HDFS 中的频率是多少？每小时？每隔 10 分钟？虽然批量处理模式能够满足一部分用户的需求，但是很多用户需要我们使用类似流水线的模式来实时采集（这样就保证了采集和后续处理之间的延迟非常小）。后一个模式，就出现了 message broker，即：以一个实时的模式从各个数据源采集数据到大数据系统上，为后续的近实时的在线分析系统和离线分析系统服务。对于这个模式，主要使用 Flume 和 Kafka 等工具。基于这些工具，一些企业实现了大数据采集平台，完成了下面的目标：

- 高性能：处理大数据的基本要求，如每秒处理几十万条数据。
- 海量式：支持 TB 级甚至是 PB 级的数据规模。
- 实时性：保证较低的延迟时间，达到秒级别，甚至是毫秒级别。
- 分布式：支持大数据的基本架构，能够平滑扩展。
- 易用性：能够快速进行开发和部署。

- 可靠性：能可靠地处理流数据。

数据采集是各种来自不同数据源的数据进入大数据系统的第一步。这个步骤的性能将会直接决定在一个给定的时间段内大数据系统能够处理的数据量的能力。数据采集过程的一些常见步骤是：解析传入数据，做必要的验证，数据清洗和数据去重，数据转换，并将其存储到某种持久层。涉及数据采集过程的逻辑步骤如图 7-1 所示。

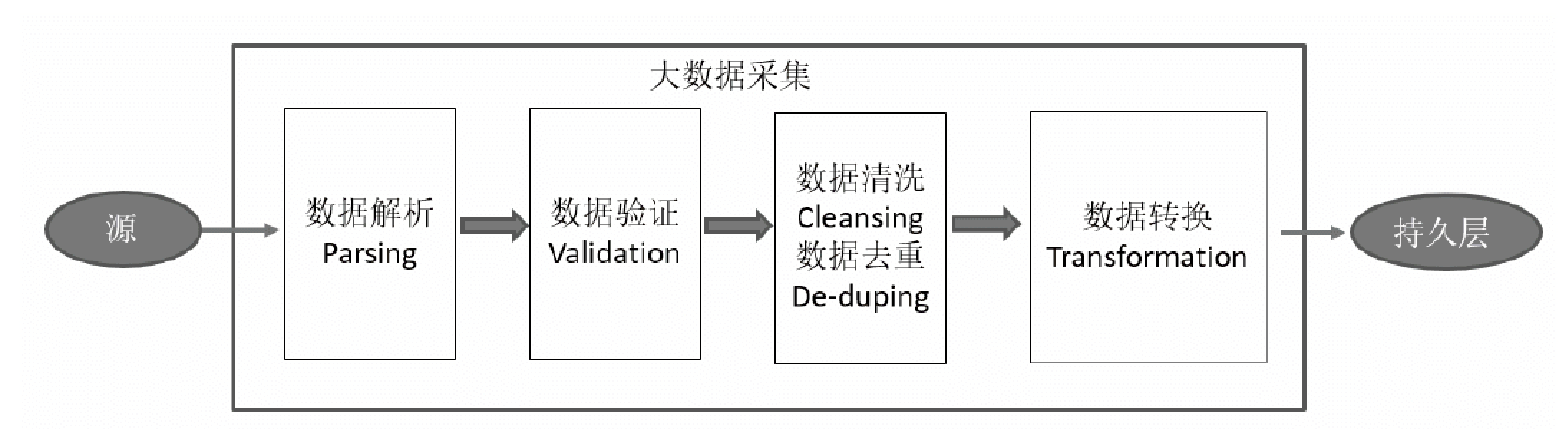


图 7-1　大数据采集步骤

采集到的大数据保存到一个持久层中，如：HDFS、HBase 等系统上。下面是一些性能方面的常用技巧：

- 来自不同数据源的传输应该是异步的。可以使用文件来传输、或者使用消息中间件来实现。由于数据异步传输，所以数据采集过程的吞吐量可以大大高于大数据系统的处理能力。异步数据传输同样可以在大数据系统和不同的数据源之间进行解耦。大数据基础架构设计使得其很容易进行动态伸缩，数据采集的峰值流量对于大数据系统来说必须是安全的。
- 如果数据是直接从外部数据库中抽取的，确保拉取数据是使用批量的方式。
- 如果数据是从文件解析，请务必使用合适的解析器。例如：如果从一个 XML 文件中读取，则有不同的解析器像 JDOM、SAX、DOM 等。类似地，对于 CSV、JSON 和其他格式的文件，也有相应的解析器和 API 可供选择。
- 优先使用成熟的验证工具。大多数解析/验证工作流程通常运行在服务器环境中。大部分的场景基本上都有现成的标准校验工具。这些标准的现成的工具一般来说要比你自己开发的工具性能要好得多。比如：如果数据是 XML 格式的，优先使用 XML（XSD）用于验证。
- 尽量提前过滤掉无效数据，以便后续的处理流程不用在无效数据上浪费过多的计算能力。处理无效数据的一个通用做法是将它们存放在一个专门的地方，这部分的数据存储占用额外的开销。
- 如果来自数据源的数据需要清洗，例如去掉一些不需要的信息，尽量保持所有数据源的抽取程序版本一致，确保一次处理的是一个大批量的数据，而不是一条记录一条记录地来处理。一般来说数据清洗需要进行数据关联。数据清洗中需要用到的静态数据关联一次，并且一次处理一个大批量数据就能够大幅提高数据处理效率。
- 来自多个源的数据可以是不同的格式。有时，需要进行数据转换，使接收到的数据从

多种格式转化成一种或一组标准格式。

一旦所有的数据采集完成后，转换后的数据通常存储在某些持久层，以便以后分析处理。有不同的持久系统，如：NoSQL 数据库、分布式文件系统等。我们要特别指出的是，数据清洗是很重要的一步。许多的数据分析最后失败，原因就是要分析的数据存在严重的质量问题，或者数据中某些因素使分析产生偏见，或使得数据科学家得出根本不存在的规律。虽然数据清洗很琐碎，但是只有事先做好了这个清洗工作，才能让分析工作卓有成效。许多初级的数据科学家往往急于求成，对数据草草处理就进行下一步分析工作，等到运行算法时，才发现数据有严重的质量问题，无法得出合理的分析结果。总之，一定要防止“垃圾进垃圾出”。

7.1 Flume

Apache Flume 是 Cloudera 提供给 Hadoop 社区的一个项目，用于从不同的数据源可靠有效地加载数据流到 HDFS 中。Flume 具有一定的容错性，并支持 failover 和系统恢复。Flume 是一个分布式、可靠、可用的轻量级工具，非常简单，容易适应各种方式的数据收集。Flume 使用 Java 编写，其需要运行在 Java 1.6 或更高版本之上。Flume 的官方网站为 http://flume.apache.org/。

7.1.1 Flume 架构

Flume 具有分布式、高可靠、高容错、易于定制和扩展的特点。它将数据从产生、传输、处理并最终写入目标的路径的过程抽象为数据流，在具体的数据流中，数据源支持在 Flume 中定制数据发送方，从而支持收集各种不同协议数据。同时，Flume 数据流提供对数据进行简单处理的能力，如过滤、格式转换等。此外，Flume 还具有能够将数据写往各种数据目标（可定制）的能力。总的来说，Flume 是一个可扩展、适合复杂环境的海量数据采集系统。Flume 的架构如图 7-2 所示。

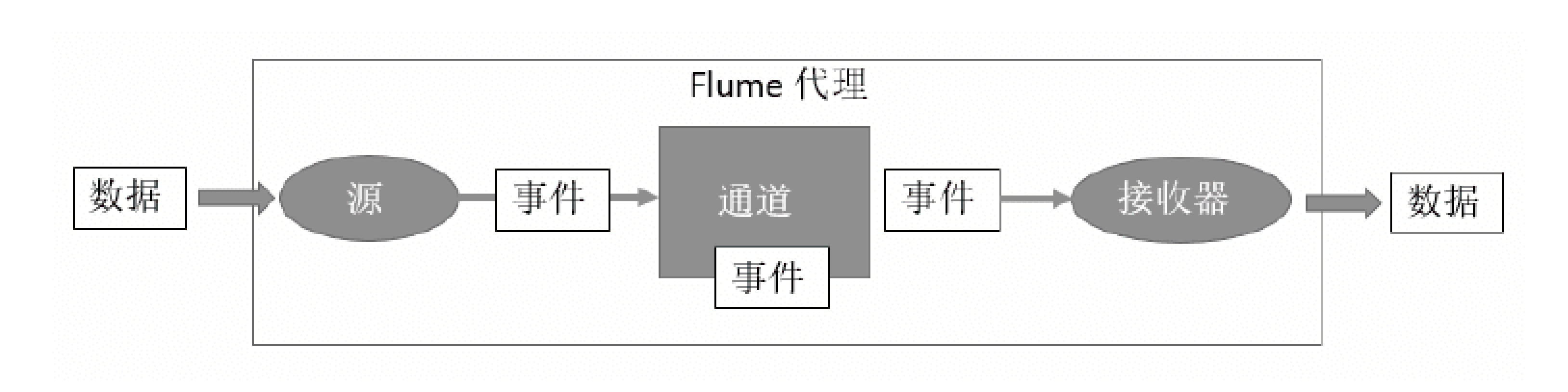


图 7-2　Flume 架构

Flume 主要有以下几个核心概念：

- 事件（Event）：Flume 传递的一个数据单元，除了采集到的数据（比如：单个日志）之外，它还带有一个可选的消息头。

- 源（Source）：输入叫做源，数据通过源（Source）进入 Flume。负责接收输入数据，并将数据写入管道，这有两种模式，一种是主动抓取数据，另一种是被动等待数据。源将事件写到一个或多个通道中。
- 接收器（Sink）：输出叫接收器，是从一个通道中接收事件，发送数据给目的地（比如：HDFS）的实体。HDFS Sink 就是一个 HDFS 文件的接收器。接收器负责从管道中读出数据并发给下一个 Agent 或者最终的目的地。
- 通道（Channel）：在 Source 和 Sink 之间传递事件的一个临时存储区，它保存有 Source 传递过来的事件（Source 把事件放到 Channel 上，Sink 从 Channel 上取出事件）。缓存从 source 到 Sink 的中间数据。可使用不同的配置来做 Channel，例如内存、文件、JDBC 等。使用内存性能高但不持久，有可能丢失数据。使用文件更可靠，但性能不如内存。
- 数据流（Flow）：事件从源点到达目的点（接收器）的迁移的抽象（就是图上细的箭头）。
- Flume 代理（Agent）：一个独立的 Flume 进程，Source、Channel 和 Sink 都运行在这个进程中。代理可能会有多个源、通道与接收器。

Flume 在 Source 和 Sink 端都使用了 transaction 机制保证在数据传输中没有数据丢失，如图 7-3 所示。

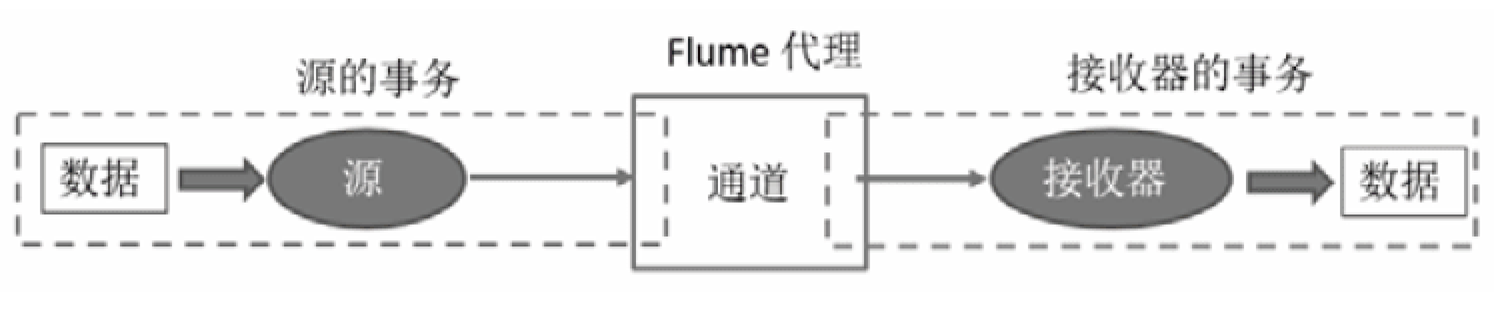


图 7-3　事务处理

7.1.2　Flume 事件

Flume 事件由 0 个或多个头与体组成，也就是说，它包含了采集的数据（“体”）和一些额外信息（“头”）的一个数据单元。Flume 事件是 Flume 传输的基本单元。头是一些键-值对（Map<String,String>），比如：事件的时间戳或发出事件的服务器主机名，类似 HTTP 头的功能。“体”是一个字节数组（byte[]），如图 7-4 所示。Flume 可能会自动添加一些头信息，比如：数据来自的主机名。



图 7-4　Flume 事件

7.1.3　Flume 源

Flume 源的类型、说明和实现的类如表 7-1 所示。

表 7-1　Flume 源的类型、说明和实现的类

类型	说明	实现的类
avro	Avro Netty RPC 事件源	AvroSource
exec	执行一个 Unix 命令，并从 stdout（标准输出）上读数据	ExecSource
netcat	监控某一个端口，从端口上读取文本行数据作为事件的数据	NetcatSource
seq	序列生成器数据源，生产序列数据	SequenceGeneratorSource
org.apache.flume.source.StressSource	压力测试使用。连续事件源，每个事件具有相同的负载。默认是 500 个字节，每个字节是一个最大为 127 的值	org.apache.flume.source.StressSource
syslogtcp	读取 syslog 数据，产生事件，支持 TCP 协议	SyslogTcpSource
syslogudp	读取 syslog 数据，产生事件，支持 UDP 协议	SyslogUDPSource
org.apache.flume.source.scribe.ScribeSource		ScribeSource

Flume 允许自定义数据源的类型和实施类，只需要通过继承 org.apache.flume.source.AbstractSource 类即可。在 Flume agent 的配置文件中，你可以定义一个或多个源。比如，下面我们定义了一个名叫 s1 的源（每个源、通道和接收器在该代理的上下文中必须要有一个唯一的名字），并指定了源的通道为 c1（注意，channels 是复数，也就是说，你可以为一个源指定多个通道），类型为 netcat，它监听 192.168.0.2 的 8083 端口上的数据：

```
agent.sources= s1
agent.channels=c1
agent.sinks=k1
agent.sources.s1.type = netcat
agent.sources.s1.channels=c1
agent.sources.s1.bind=192.168.0.2
agent.sources.s1.port=8083
```

在安装完 Flume，配置上面的参数之后，就可以启动它了。这时候，源就在 192.168.0.2 的

8083 端口上进行监听。你可以使用 nc 命令（或其他网络客户端）给上述端口发送数据，这些数据将被写到内存通道中，然后被送到接收器上。因为接收器是 log4j 日志文件，你可以打开日志文件来确认是否收到了数据。

Flume 有一个 exec 源，它提供了在 Flume 外执行命令，并将输出结果转换为 Flume 事件的机制。比如：下面的配置将会对/yunsheng/log/huanbao.log 文件执行 tail 命令，把日志数据写到 c2 通道上。

```
agent.sources= s2
agent.channels=c2
agent.sinks=k2
agent.sources.s2.type =exec
agent.sources.s2.channels=c2
agent.sources.s2.command=tail -F /yunsheng/log/huanbao.log
```

7.1.4　Flume 拦截器（Interceptor）

数据采集的理想做法是：由数据的生产者在把数据发送到平台之前对数据进行清理。这应当由产生数据的团队来处理，因为他们最了解他们自己的数据。如图 7-5 所示，拦截器是数据流中的一个处理点，它可以在源和通道之间插入一个或多个拦截器，来动态检查和修改 Flume 事件。有点类似 Servlet 的 ServletFilter。

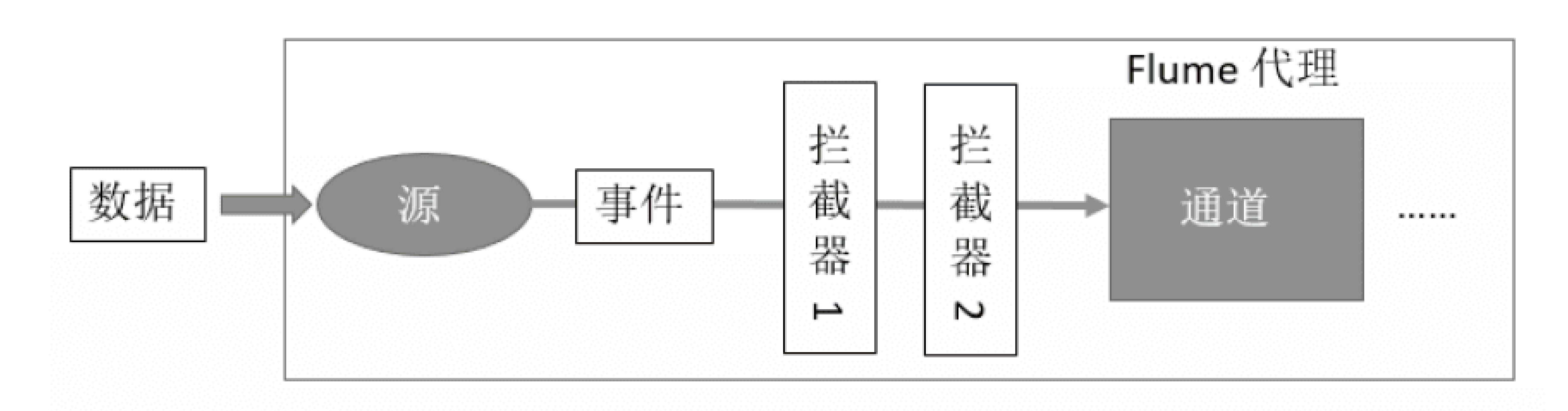


图 7-5　拦截器

下面这个例子添加了 4 个拦截器（i1、i2、i3 和 i4，i2 接收 i1 的处理结果，i3 接收 i2 的处理结果，i4 接收 i3 的处理结果，并将结果送给通道选择器）。其中 i1 是一个时间戳拦截器，如果源中没有 timestamp 头，那么，拦截器会添加之。i2 是一个 Host 拦截器，它会向事件中添加一个 Flume 代理所在的 IP 的头。i3 是一个 static 拦截器，可用于添加一对“键值”。i4 是一个正则表达式过滤拦截器，这会根据“体”（即：传输数据）的内容来过滤事件。在 regex 上设置模式字符串，如果你设置了 excludeEvents 的值为 false（默认值），那么，只保留与模式匹配的事件；如果为 true，那么，过滤掉匹配的事件。

```
agent.sources.s1.interceptors= i1 i2 i3 i4
agent.sources.s1.interceptors.i1.type=timestamp
agent.sources.s1.interceptors.i1.perserveExisting=true
agent.sources.s1.interceptors.i2.type=host
agent.sources.s1.interceptors.i3.type=static
agent.sources.s1.interceptors.i3.key=键
```

```
agent.sources.s1.interceptors.i3.value=值
agent.sources.s1.interceptors.i4.type=regex_filter
agent.sources.s1.interceptors.i4.regex=测试数据
agent.sources.s1.interceptors.i4.excludeEvents=true
```

上面的 perserveExisting 指定是保留头上的相关信息，还是覆盖之。还有一个拦截器是正则表达式抽取过滤器。它可以抽取事件“体”（即：数据）的内容，并放到 Flume 头上，以便通道选择器根据这些值路由不同的通道。

Flume 还允许我们自己定义拦截器，这就需要实现 org.apache.flume.interceptor.Interceptor 和 org.apache.flume.interceptor.Interceptor.Builder 接口。假定你的类名为 com.yunsheng.Test。那么，你可以设置为：

```
agent.sources.s1.interceptors =i5
agent.sources.s1.interceptors.i5.type=com.yunsheng.Test$Builder
```

7.1.5　Flume 通道选择器（Channel Selector）

如图 7-6 所示，Source 上的数据可以复制到不同的通道上。每一个 Channel 也可以连接不同数量的 Sink。这样连接不同配置的 Agent 就可以组成一个复杂的数据收集网络。通过对 Agent 的配置，可以组成一个路由复杂的数据传输网络。

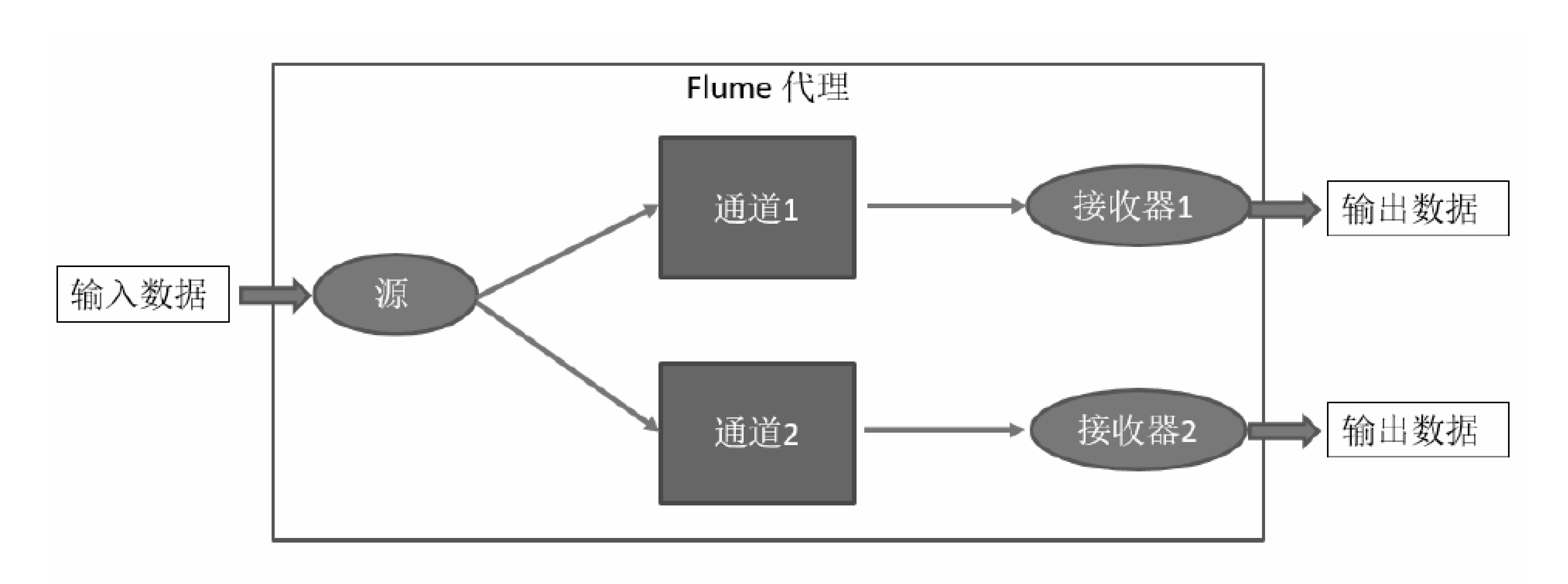


图 7-6　通道选择器

通道选择器负责将数据从一个源转到一个或多个通道上。Flume 提供了 Replicating Channel Selector（复制通道选择器）、Multiplexing Channel Selector（多路通道选择器）和 Custom Channel Selector（自定义通道选择器）。复制通道选择器是一个默认的通道选择器，负责将事件复制到每个通道上，而多路通道选择器会根据某些头信息把事件放到不同的通道上。通道选择器和拦截器一起构成了具有简单的工作流功能的多通道路由。

下面这个例子设置了多路通道选择器，并设置了通道选择器使用头信息 port 的值来发送不同端口来的数据到不同的通道上。8081 端口的数据到 c1 通道，8082 端口的数据到 c2 通道，而 8083 端口的数据到 c3 通道。

```
agent.sources.s1.channels= c1 c2 c3
```

```
agent.sources.s1.selector.type=multiplexing
agent.sources.s1.selector.header=port
agent.sources.s1.selector.default=c1
agent.sources.s1.selector.mapping.8081=c1
agent.sources.s1.selector.mapping.8082=c2
agent.sources.s1.selector.mapping.8083=c3
```

7.1.6 Flume 通道

在 Flume 中，通道指的是位于源与接收器之间的组件，为流动的事件数据提供了一个中间缓存区域。Flume 通道的类型、说明和实现的类如表 7-2 所示。

表 7-2 Flume 通道的类型、说明和实现的类

类型	说明	实现的类
memory	事件数据存储在内存中，这是最快的，但是不持久的事件传输	MemoryChannel
file	事件数据存储在磁盘文件中	FileChannel
jdbc	事件数据存储在基于 JDBC 的可持久化的数据库中（目前 Flume 内置支持 Derby）	JDBCChannel
recoverablememory	一个可持久化的通道，使用本地文件系统作为存储	RecoverableMemoryChannel
org.apache.flume.channel.PseudoTxnMemoryChannel	测试用	PseudoTxnMemoryChannel

对于上面的 memory 和 recoverablemeory 两个类型，recoverablememory 是本地文件系统的持久化通道，它要比 memory 慢一点，因为它需要在接收器接到数据之前将所有变化写到本地文件系统上（即：磁盘上）。当出现问题（如：硬件问题、JVM 崩溃等）而 Flume 代理需要重启时，这些事件数据可以被恢复。与 memory 相比，recoverablememory 可以缓存更多的事件数据。至于选择哪个类型，这取决于用户的实际需要。理论上说，如果从源到通道的数据存储率大于接收器从通道获取数据的速率，那么，就会超出通道的处理能力，会抛出 ChannelException 异常。所以，源需要对这个异常做一些处理。

Flume 还允许自定义通道的类型和实施类。和源一样，在 Flume agent 的配置文件中，你可以定义一个或多个通道，比如，下面我们定义了一个名叫 c1 的通道（每个通道在该代理的上下文中必须要有一个唯一的名字），并指定了通道能够持有的最大的事件数量为 200（默认为 100。如果增加了这个值，你可能需要增加 JVM 堆空间的大小），它监听 192.168.0.2 的 8083 端口上的数据：

```
agent.sources= s1
agent.channels=c1
```

```
agent.sinks=k1
agent.sources.s1.type = netcat
agent.sources.s1.channels=c1
agent.sources.s1.bind=192.168.0.2
agent.sources.s1.port=8083
agent.channels.c1.type=memory
agent.channels.c1.capacity=200
```

Flume 通道还有其他一些配置，比如：transactionCapacity 是指定源的 ChannelProcessor（负责单个事务中将数据从源移动到通道中的组件）可以写入的最大的事件数量，也指的是 SinkProcessor（负责将数据从通道移动到接收器的组件）在单个事务中所能读取的最大的事件的数量。如果接收器能够进行大数据量的处理，那么，加大这个值会提高速度。还有一个参数是 keep-alive，它指的是：当通道已经满了，等待写入数据的时间。其他的参数如 byteCapacity 和 BufferPercentage 的设置是同 JAVA 的 OutOfMemoryErrors 有关。如果得到了 OutOfMemoryErrors 的错误，可以考虑调整这些参数。

对于 file 类型的 Flume 通道，你可以设置 checkpointDir 和 dataDirs 属性来为不同的通道指定不同的目录。默认的文件通道的容量是 100 万个事件。

7.1.7　Flume 接收器

Flume 接收器支持的类型、说明和实现类如表 7-3 所示。

表 7-3　Flume 接收器支持的类型、说明和实现的类

类型	说明	实现类
hdfs	数据写入 HDFS	HDFSEventSink
org.apache.flume.sink.hbase.HBaseSink 或 org.apache.flume.sink.hbase.AsyncHBaseSink	数据写入 HBase 数据库	org.apache.flume.sink.hbase.HBaseSink 或 org.apache.flume.sink.hbase.AsyncHBaseSink
logger	数据写入日志文件（默认为 log4j，INFO 级别，可配置）	LoggerSink
avro	数据被转换成 avro 事件，然后发送到配置的 RPC 端口上	AvroSink
file_roll	存储数据到本地文件系统上	RollingFileSink
irc	数据在 IRC 上	IRCSink
null	丢弃所有数据（/dev/null）	NullSink

Flume 还允许自定义接收器的类型和实施类，只需要通过继承 org.apache.flume.sink.AbstractSink 类即可。和源和通道一样，在 Flume agent 的配置文件中，你可以定义一个或多个接收器，比如，下面我们定义了一个名叫 k1 的接收器（每个接收器在该代理的上下文中必须要有一个唯一的名字），并指定了接收器的类型为 logger（该类型的接收器主要用于调试与测试，它默认使用 log4j 将所有 INFO 级别的日志记录下来），接收器 k1 的数据来自于通道 c1。“agent.sinks.k1.channel”上的 channel 是单数，这是因为一个接收器只能从一个通道接收数据：

```
agent.sources= s1
agent.channels=c1
agent.sinks=k1
agent.sources.s1.type = netcat
agent.sources.s1.channels=c1
agent.sources.s1.bind=192.168.0.2
agent.sources.s1.port=8083
agent.channels.c1.type=memory
agent.channels.c1.capacity=200
agent.sinks.k1.type=logger
agent.sinks.k1.channel=c1
```

下面我们来看一下 HDFS 接收器。HDFS 接收器的作用是持续打开 HDFS 中的文件，然后以流的方式将数据写入其中。为了使用 HDFS 接收器，可将接收器的类型设置为 hdfs，并设置 HDFS 路径：

```
agent.sinks.k2.type=hdfs
agent.sinks.k2.hdfs.path=/path/in/hdfs
agent.sinks.k2.channel=c2
```

对于 HDFS 路径值，Flume 支持基于时间的转义序列，比如：

```
agent.sinks.k2.hdfs.path=/path/in/hdfs/%Y/%m/%D/%H
```

也可以使用事件的头值（假定键为 logType 的头）来将数据写到不同的 HDFS 路径，比如：

```
agent.sinks.k2.hdfs.path=/path/in/hdfs/%{logType}/%Y/%m/%D/%H
```

除了 path，还可以设置文件名的前缀（.filePrefix）和后缀（.fileSuffix）。Flume 还支持压缩文件，比如：

```
agent.sinks.k2.hdfs.codeC=gzip
```

上面是设置输出文件的路径和文件名。对于数据输出，可以使用序列器，比如：下面我们采用 text 序列化器（默认设置），它只会输出 Flume 事件体（即：只是数据本身），而丢弃头信

息。因为“appendNewLink=true”，所以在文件中的每个事件数据后面都有一个换行符。

```
agent.sinks.k2.serializer=text
agent.sinks.k2.serializer.appendNewLink=true
```

如果需要带有头信息的文本，可选用 text_with_headers。

7.1.8　负载均衡和单点失败

为了解决 Flume 接收器的单点失败的问题，Flume 支持接收器组的概念，通过负载平衡将事件发送到不同的接收器。下面我们设置了一个名叫 sg 的接收器组，这个组包含了四个接收器 k1、k2、k3 和 k4：

```
agent.sinkgroups=sg
agent.sinkgroups.sg.sinks=k1,k2,k3,k4
agent.sinkgroups.sg.processor.type=failover
agent.sinkgroups.sg.processor.priority.k1=10
agent.sinkgroups.sg.processor.priority.k2=20
agent.sinkgroups.sg.processor.priority.k2=30
agent.sinkgroups.sg.processor.priority.k2=40
```

如果将上述的 processor.type 设置为 load_balance，那么，就会使用循环模式来均衡地对这四个接收器进行流量的负载。对于 failover 设置，就是指当某个接收器不能用时，通过 priority 属性来指定优先顺序。在上述的例子中，先尝试 k1，然后 k2，以此类推。

7.1.9　Flume 监控管理

我们首先要对 Flume 代理进程进行监控，监测 Flume 代理是否正常运行。如果已经停止，就需要重启。有很多工具，如：Monit 和 Nagios 就是不错的开源免费监控工具。对于 Flume 内部的监控，我们可以使用 Ganglia 工具，它是一个开源的监控工具。在 Flume 启动时，配置几个属性值就可以让 Flume 向 Ganglia 工具发送监测数据：

```
flume.monitoring.type=ganglia
flume.monitoring.hosts=host1:port1, host2:port2
flume.monitoring.pollInterval=60
```

最后一个参数设置发送数据给 Ganglia 的时间间隔（单位为秒）。另一种方法是，启动 Flume 内部 HTTP 服务器，然后从外部通过 HTTP 请求查询 Flume 的状态（返回结果为 JSON）。如果要使用这个方式，则在启动时设置如下属性：

```
flume.monitoring.type=http
```

```
flume.monitoring.port=9088
```

这样的话，就可以使用 http://hostname:9088/metrics 来获得 JSON 数据。

7.1.10 Flume 实例

下面我们看一个例子。某路由器厂商想要分析各个家庭路由器的数据。由于这个厂商是给我们开放了他们的 FTP 服务器（他们自己把路由器数据首先采集到一个 FTP 服务器上），所以，我们是把 FTPserver 作为数据源获取数据的，数据以小文件的方式存储在 Flume 所监控的目录池下，文件会自动被 Flume 读取并删除，读取的小文件会直接送到 HDFS。下面是我们采集的具体步骤：

步骤 01 确保 Flume 已经安装成功。在界面上查看状态显示为绿色，如图 7-7 所示。



图 7-7 界面上查看状态显示为绿色

步骤 02 创建 Flume 的目录池，在/usr/hdp/2.3.0.0-2557/flume/conf 下创建一个目录。

步骤 03 配置 Flume 如下：

```
# Flume agent config

agent1.sources=source1
agent1.sinks=sink1
agent1.channels=channel1


#source1
agent1.sources.source1.type=spooldir
agent1.sources.source1.spoolDir=/usr/hdp/2.3.0.0-
2557/flume/conf/test1
agent1.sources.source1.channels=channel1
agent1.sources.source1.fileHeader = true
agent1.sources.source1.deletePolicy = immediate
agent1.sources.source1.batchSize=1000
agent1.sources.source1.batchDurationMillis=1000
agent1.sources.source1.decodeErrorPolicy = IGNORE
agent1.sources.source1.interceptors=i1
agent1.sources.source1.interceptors.i1.type=timestamp
```

```
#sink1
agent1.sinks.sink1.type=hdfs
agent1.sinks.sink1.hdfs.fileType=DataStream
agent1.sinks.sink1.hdfs.writeFormat=TEXT
agent1.sinks.sink1.channel=channel1
agent1.sinks.sink1.hdfs.path =hdfs://master01:8020/flume3/%Y-%m-%d
agent1.sinks.sink1.hdfs.filePrefix = PhicommSOHO-logFile.%Y-%m-%d-%h
agent1.sinks.sink1.hdfs.inUsePrefix =.
agent1.sinks.sink1.hdfs.idleTimeout=0
agent1.sinks.sink1.hdfs.useLocalTimeStamp = true
agent1.sinks.sink1.hdfs.rollInterval=3600
agent1.sinks.sink1.hdfs.rollSize=128000000
agent1.sinks.sink1.hdfs.rollCount=0
agent1.sinks.sink1.hdfs.batchSize = 1000

#channel1
agent1.channels.channel1.type=memory
agent1.channels.channel1.checkpointDir=/usr/hdp/2.3.0.0-
2557/flume/conf/testchannelcheck1
agent1.channels.channel1.dataDirs=/usr/hdp/2.3.0.0-
2557/flume/conf/testchannel1
```

步骤 04　启动 Flume，并执行以下命令：

```
flume-ng agent -n  agent3 -c conf -f /usr/hdp/2.3.0.0-
2557/flume/conf/agent3/flume.conf - Dflume.root.logger=DEBUG,console
```

关于 Flume 的更多内容，可参考以下网站：

- Flume 官方网站：http://flume.apache.org/。
- Flume 用户文档：http://flume.apache.org/FlumeUserGuide.html。
- Flume 开发文档：http://flume.apache.org/FlumeDeveloperGuide.html。

7.2 Kafka

Kafka 是 2010 年 12 月开源的项目，采用 scala 语言编写。Kafka 是一个分布式的消息队列，可用在不同的系统之间传递数据。如图 7-8 所示，如果各部门间的数据交换都是交换双方之间独

自建设的交换通道。随着交换部门的增多，每个部门要建设和维护的交换通道也增多，多条交换通道交叉互联，最后形成一张维护难度极大的数据交换网络，显然，这种数据交换模式难以满足大数据交换和共享需求。

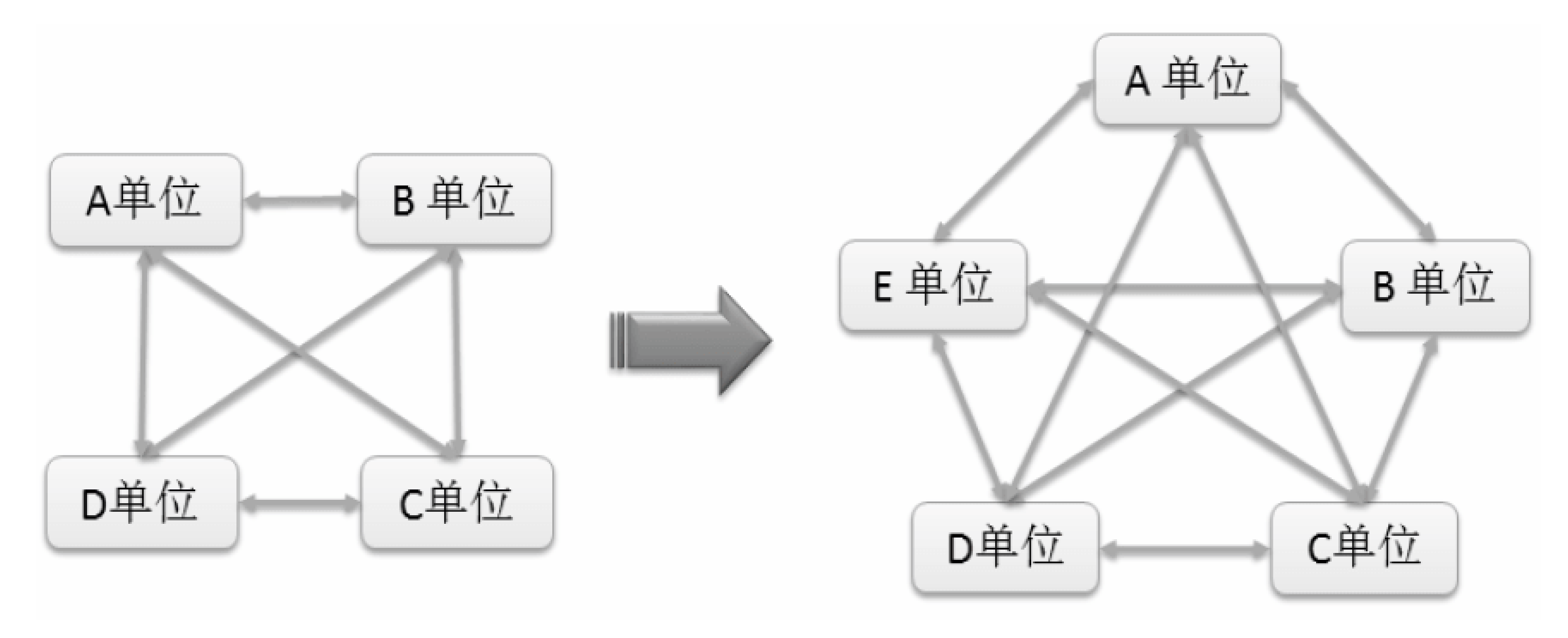


图 7-8　交换网络扩张示意图

Kafka 就可以解决上述问题，它可以担当中间桥梁。如图 7-9 所示，它使用了多种优化机制，整体架构比较新颖（push/pull），更适合异构集群。它实现了高吞吐率，在普通的服务器上每秒也能处理几十万条消息。比如：LinkedIn 公司每天通过 Kafka 运行着超过 600 亿个不同的消息写入点。

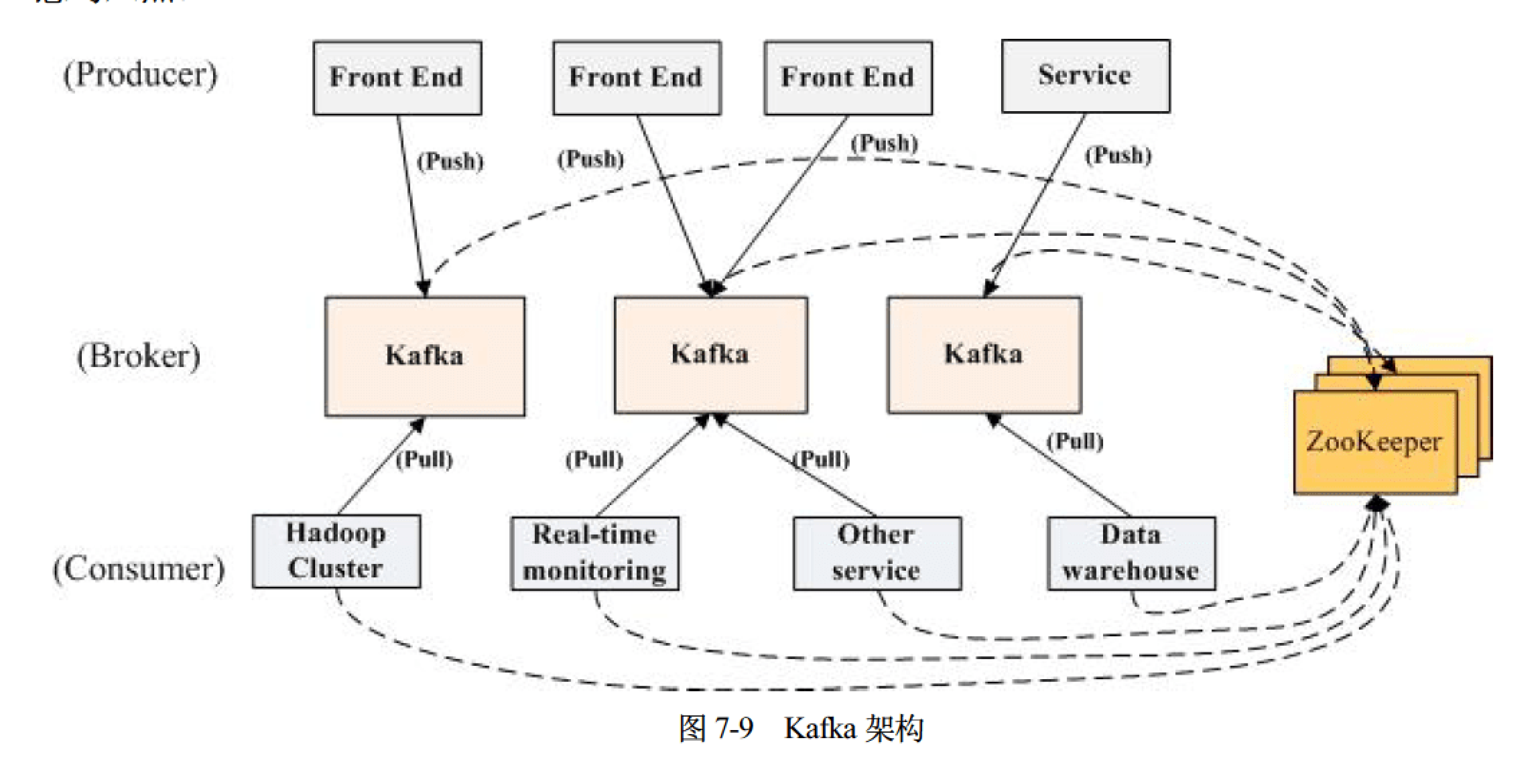


图 7-9　Kafka 架构

7.2.1　Kafka 架构

Kafka 提供了类似于 JMS 的特性，但是在设计实现上完全不同（它并不是 JMS 规范的实现）。在消息保存时，Kafka 根据 Topic 进行归类，发送消息者成为 Producer，消息接受者成为 Consumer。此外，Kafka 集群由多个 Kafka 实例组成，每个实例（server）成为 Broker。无论是 Kafka 集群，还是 Producer 和 Consumer 都依赖于 Zookeeper 来保证分布式协作。每条发布到

Kafka 集群的消息都有一个类别，这个类别被称为 Topic。物理上不同 Topic 的消息分开存储，逻辑上一个 Topic 的消息虽然保存于一个或多个 Broker 上，但用户只需指定消息的 Topic，即可生产或消费数据而不必关心数据存于何处。

Kafka 实际上是一个消息发布订阅系统。Producer（生产者）向某个 Topic 发布消息，而 Consumer（消费者）订阅某个 Topic 的消息，进而一旦有新的关于某个 Topic 的消息，Broker 会传递给订阅它的所有 Consumer。在 Kafka 中，消息是按 Topic 组织的，而每个 Topic 又分为多个 partition，这样便于管理数据和进行负载均衡。同时，它也使用了 Zookeeper 进行负载均衡。Kafka 中主要有三种角色，分别为 Producer、Broker 和 Consumer。

（1）Producer（生产者）

Producer 用于将流数据发送到 Kafka 消息队列上，它的任务是向 Broker 发送数据。Kafka 提供了两种 Producer 接口，一种是 low level 接口，使用该接口会向特定的 Broker 的某个 Topic 下的某个 partition 发送数据；另一种是 high level 接口，该接口支持同步/异步发送数据，基于 Zookeeper 的 Broker 自动识别和负载均衡（基于 Partitioner）。Producer 可以通过 Zookeeper 获取可用的 Broker 列表，也可以在 Zookeeper 中注册 listener，该 listener 在以下情况下会被唤醒：

- 添加一个 Broker。
- 删除一个 Broker。
- 注册新的 Topic。
- Broker 注册已存在的 Topic。

当 Producer 得知以上事件时，可根据需要采取一定的行动。以日志采集为例，生产过程分为三部分：第一部分为监控日志采集的本地日志文件或目录，如果有变化，则将变化的内容逐行读取到内存的消息队列中。第二部分是连接 Kafka 集群，这包括一些配置信息，比如：是否压缩、超时等。第三部分是用于将已经获取的数据通过上述的连接，发送到 Kafka 集群中，是一个推送的过程。

（2）Broker

Kafka 集群包含一个或多个服务器，这种服务器被称为 Broker。Broker 采取了多种策略提高数据处理效率，包括 sendfile 和 zero copy 等技术。

（3）Consumer（消费者）

Consumer 的作用是处理数据，比如：将信息加载到持久化存储系统上。Kafka 提供了两种 Consumer 接口，一种是 low level 的，它维护到某一个 Broker 的连接，并且这个连接是无状态的，即每次从 Broker 上 pull 数据时，都要告诉 Broker 数据的偏移量。另一种是 high level 接口，它隐藏了 Broker 的细节，允许 Consumer 从 Broker 上 pull 数据而不必关心网络拓扑结构。

7.2.2 Kafka 与 JMS 的异同

kafka 与 JMS（Java Message Service）实现的不同之处是，即使消息被消费，消息不会被立即删除。它将会根据 Broker 中的配置要求，保留一定的时间之后才删除。比如 log 文件保留 2 天,那么两天后，文件会被清除，无论其中的消息是否被消费。Kafka 通过这种简单的方式来释放磁盘空间，以减少消息消费之后产生太多的磁盘 I/O。在 JMS 实现中，Topic 模型基于 push 方式，即 Broker 将消息推送给 Consumer 端。而在 Kafka 中，则采用了 pull 方式，即 Consumer 在和 Broker 建立连接之后，主动去 pull（或者说 fetch）消息。这种模式有些优点，首先 Consumer 端可以根据自己的消费能力适时地去获取消息并处理，且可以控制消息消费的进度（offset）。此外，消费者可以良好地控制消息消费的数量。

7.2.3 Kafka 性能考虑

Kafka 集群几乎不需要维护任何 Consumer 和 Producer 状态信息，这些信息有 Zookeeper 保存。因此 Producer 和 Consumer 的客户端实现非常轻量级，它们可以随意离开，而不会对集群造成额外的影响。

Kafka 是基于文件存储。通过分区（partiton），可以将文件内容分散到多个 server 上来避免文件大小达到单机磁盘的上限，每个分区都会被当前 server（Kafka 实例）保存。我们可以将一个 Topic 切分为多个分区来提高消息保存/消费的效率。此外，越多的分区意味着可以容纳更多的 Consumer，有效提升并发消费的能力。

对于一些常规的消息系统，Kafka 是个不错的选择。partitons/replication 和容错功能可以使 Kafka 具有良好的扩展性和性能优势。影响 Kafka 性能的因素很多，除磁盘 I/O 之外，我们还需要考虑网络 I/O，这直接关系到 Kafka 的吞吐量问题。对于 Producer 端，可以将消息 buffer 起来。当消息的条数达到一定阀值时，批量发送给 Broker；对于 Consumer 端也是一样，批量抓取多条消息，消息量的大小可以通过配置文件来指定。对于 Kafka Broker 端，将文件的数据映射到系统内存中，socket 直接读取相应的内存区域即可，而无须再次拷贝和交换。对于 Producer/Consumer/Broker 三者而言，CPU 的开支应该都不大，因此启用消息压缩机制是一个良好的策略。压缩需要消耗少量的 CPU 资源，但减少了网络 I/O 的开销。Kafka 支持 gzip/snappy 等多种压缩方式。

生产者采用了负载均衡：Producer 将会和 Topic 下所有 partition leader 保持 socket 连接；消息由 Producer 直接通过 socket 发送到 Broker，中间不会经过任何路由层。事实上，消息被路由到哪个分区上，有 Producer 客户端决定。如果一个 Topic 中有多个分区，那么在 Producer 端实现“消息均衡分发”是必要的。其中 partition leader 的位置（host:port）注册在 Zookeeper 中，Producer 作为 Zookeeper client，已经注册了 watch 用来监听 partition leader 的变更事件。我们也可以采用异步发送，即：将多条消息暂且在客户端 buffer 起来，并将他们批量发送到 Broker，小数据 I/O 太多，会拖慢整体的网络延迟，批量延迟发送事实上提升了网络效率。不过这也存在一定的隐患，比如说，当 Producer 失效时，那些尚未发送的消息将会丢失。

7.2.4　消息传送机制

对于 JMS 实现，消息传输机制只有一种：有且只有一次（exactly once）。在 Kafka 中稍有不同：

- at most once：最多一次，这个和 JMS 中"非持久化"消息类似。发送一次，无论成败，将不会重发。
- at least once：消息至少发送一次，如果消息未能接受成功，可能会重发，直到接收成功。
- exactly once：消息只会发送一次。

在上面的 at most once 模式中，消费者获取消息，保存 offset，然后处理消息。当 client 保存 offset 之后，在消息处理过程中出现了异常，导致部分消息未能继续处理。那么此后"未处理"的消息将不能被获取到，这就是"at most once"。而"at least once"是消费者获取消息，处理消息，然后保存 offset。在消息处理成功之后，但在保存 offset 阶段 Zookeeper 异常而导致保存操作未能执行成功，这就导致接下来再次获取时，可能获得上次已经处理过的消息，这就是"at least once"。原因是 offset 没有及时地提交给 Zookeeper，Zookeeper 恢复正常后还是之前 offset 状态。"exactly once"在 Kafka 中并没有严格地去实现，这种策略在 Kafka 中可能是不需要的。

通常情况下，"at least once"是我们的首选。相比"at most once"而言，重复接收数据总比丢失数据要好。

7.2.5　Kafka 和 Flume 的比较

Kafka 和 Flume 这两个产品的功能有些重叠，但是也有一些区别：

- Kafka 是一个更加通用的系统。你可以有很多数据的 Producer 和数据的 Consumer。这些 Consumer 之间共享多个主题。而 Flume 主要是为了发送数据给 HDFS 和 HBase 用的工具。Flume 集成了 Hadoop 的安全体系。Cloudera 认为，如果数据将被多个系统所消费，那么采用 Kafka。
- Flume 具有多个内置的源和 sink，相对而言，Kafka 只有一个较少的 Producer 和 Consumer 生态圈。所以，如果 Flume 的源和 sink 正好满足你的要求，而且你希望使用一个不需要开发的采集系统（只需要配置），那么，你就使用 Flume。如果你自己想开发一个采集系统，那就基于 Kafka 开发。
- Flume 可以使用 interceptors 来即时处理数据，这会对数据过滤有帮助；Kafka 需要外部的流处理系统来完成这个功能。
- Flume 和 Kafka 都是可靠的系统，都可以保证数据不会丢失。但是，Flume 不复制事件。因此，你即使使用了可靠的文件 channel，如果一个 Flume 代理的节点崩溃，你就无法访问这个节点上的事件（直到你恢复了这个硬盘的访问）。如果你需要一个很强的高可用性的输入管道，那么，优先使用 Kafka。
- Flume 和 Kafka 可以一起工作。如果你需要从 Kafka 流数据到 Hadoop，那么，你可以将 Flume 代理和 Kafka 源一起使用来读取数据。这样的话，你就不需要实现你自己的

Consumer 了，你获得了所有与 HDFS 和 HBase 完美集成的 Flume 的好处了。

7.3 Sqoop

Sqoop 是 SQL to Hadoop 的缩写，是一个数据库导入导出工具，可以将数据从 Hadoop 导入到关系数据库，或从关系数据库将数据导入到 Hadoop 中。Sqoop 使用了 MapReduce 来导入导出数据，充分利用了 MapReduce 的并行性和容错性。Sqoop 主要是在 Linux 环境下使用。你可以直接运行 Sqoop 命令：

```
$ sqoop help
usage: sqoop COMMAND [ARGS]

Available commands:
  codegen            Generate code to interact with database records
  create-hive-table  Import a table definition into Hive
  eval               Evaluate a SQL statement and display the results
  export             Export an HDFS directory to a database table
  help               List available commands
  import             Import a table from a database to HDFS
  import-all-tables  Import tables from a database to HDFS
  import-mainframe   Import mainframe datasets to HDFS
  list-databases     List available databases on a server
  list-tables        List available tables in a database
  version            Display version information

See 'sqoop help COMMAND' for information on a specific command.
```

7.3.1 从数据库导入 HDFS

Sqoop 支持从关系型数据库导入表数据到 HDFS 的文件上。导入过程可以是并行的，可以产生多个文件，每个文件包含一些表数据。下面我们看几个例子。

（1）在 MySQL 创建表

```
create table test (clientmac VARCHAR(64), time VARCHAR(64), content
VARCHAR(64));
```

（2）给 MySQL 表装载一些测试数据

```
    load data local infile '/usr/hdp/2.2.0.0-2041/flume/30.txt' into
table test columns terminated by ' ' lines terminated by '\r\n';
```

（3）将表数据导入到 HDFS 文件中，一行表数据为文件的一行

```
    sqoop import -m 1 --connect jdbc:mysql://192.168.1.107:3306/test --
table test --target-dir /user/test
```

在上述命令中，-m 参数决定了将会启动多少个 mapper 来执行数据导入（只有一个“-”的参数叫做通用参数）。因为上述例子将-m 设置为 1，所以就启动了 1 个 mapper 用于导入数据。每个 mapper 将产生一个独立的文件。--connect 参数定义了 JDBC 驱动（有两个“-”的参数叫做工具相关参数）以用来连接数据库，--table 参数告诉了 Sqoop 哪个表的数据需要被导出，--target-dir 参数决定了导出的表数据将被存储在 HDFS 的哪个目录下。导出的文件可以是一个包含了定界符的文本文件，也可以通过设置其他参数来指定数据文件的格式为 avro 文件或序列化的文件。

你可以将上述的常用参数和参数值放到一个文件中，这样就省得每次都把一堆命令敲在 Sqoop 后面了。比如：

```
    sqoop -m 1 -options-file /users/zhenghong/import.txt --table test --
target-dir /user/test
```

而在 import.txt 文件中，我们有如下设置：

```
    import
    --connect
    jdbc:mysql://192.168.1.107:3306/test
```

Sqoop 还提供了 sqoop-job 的功能。与上面的例子不同，我们可以把常用的配置信息在 Sqoop 上以 job 的形式保存（而不是上面的文件）。比如：

```
    sqoop job --create myjob -- import --connect
jdbc:mysql://example.com/db --table mytable
```

然后可以执行这个保存的 job：

```
    sqoop job --exec myjob
```

Sqoop 支持对表的部分数据导入到 HDFS 文件上。下面表 7-4 列出了 import 的常用参数。

表 7-4　import 的常用参数

参数	说明
--append	附加数据到 HDFS 的一个已经存在的数据集上
--as-avrodatafile	导入数据到 Avro Data Files
--as-sequencefile	导入数据到 SequenceFiles，这是一个二进制格式的序列化的文件
--as-textfile	导入数据为一个文本文件（默认值）。一行数据为文件的一行，每个列之间有定界符分隔开。分隔符包括逗号、tab 或其他的
--as-parquetfile	导入数据到 Parquet 文件
--boundary-query <statement>	边界查询，用于创建分割
--columns <col,col,col…>	指定导入表上的哪些列
--delete-target-dir	如果导入的目标目录已经存在，则删除之
--direct	直接使用数据库支持的本地导入导出工具。比如：MySQL 自己提供了一个 mysqldump 工具，该工具比 JDBC 连接的方式快。这个参数可以直接使用这个工具
--fetch-size <n>	指定一次从数据库读取的行数
--inline-lob-limit <n>	对一个 inline LOB 设置最大大小
-m,--num-mappers <n>	决定了将会启动多少个 mapper 来执行数据导入，每个 mapper 将产生一个独立的文件
-e,--query <statement>	指定查询语句
--split-by <column-name>	指定一个表的列，这个列用于分割工作单元。比如：指定了 ID 列，该 ID 有 0~1000 的数值，我们又指定了-m 4，那么，并行执行的导入进程将分别处理（0,250）、（250,500）、（500,750）和（750,1000）
--autoreset-to-one-mapper	如果这个表没有主键，而且没提供 split-by 列，则导入命令只使用一个 mapper
--table <table-name>	指定表
--target-dir <dir>	导出的表数据将被存储在 HDFS 的哪个目录下
--warehouse-dir <dir>	指定 HDFS 文件目录的父目录
--where <where clause>	WHERE 语句
-z,--compress	启用压缩，即：在导入的过程中对数据进行压缩，默认的压缩方式为 GZIP 压缩
--compression-codec <c>	使用 Hadoop 支持的压缩编码（默认为 gzip）
--null-string <null-string>	对于字符串列，用该字符串来替代 null 值
--null-non-string <null-string>	对于非字符串列，用该字符串来替代 null 值

比如，下面这个例子就是使用一个 SELECT 语句来指定想要导入的数据：

```
sqoop eval --connect jdbc:mysql://192.168.2.100:3306/hive --username
```

```
hive --password password --query "select * from dadian where
createTime='123'";
```

执行结果如图 7-10 所示。

```
16/02/19 17:17:54 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6.2.3.2.0-2950
16/02/19 17:17:54 WARN tool.BaseSqoopTool: Setting your password on the command-line is insecure. Consider using -P instead.
16/02/19 17:17:54 INFO manager.MySQLManager: Preparing to use a MySQL streaming resultset.
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/2.3.2.0-2950/hadoop/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/2.3.2.0-2950/zookeeper/lib/slf4j-log4j12-1.6.1.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
---------------------------------------------------------------------------
| WANMAC          | setType         | deviceType          | createTime          |
---------------------------------------------------------------------------
| a               | b               | c                   | 123                 |
| aa              | bb              | cc                  | 123                 |
| aa              | bb              | cc                  | 123                 |
| aa              | bb              | cc                  | 123                 |
| aa              | bb              | cc                  | 123                 |
---------------------------------------------------------------------------
```

图 7-10　导入的数据命令执行结果

7.3.2　增量导入

Sqoop 提供了增量导入的功能，能够从上次导入的点上导入新增的数据。下面是同增量导入相关的参数，如表 7-5 所示。

表 7-5　同增量导入相关的参数

参数	说明
--check-column (col)	指定要检查的列（以这个列值为基础确定哪些列要导入）。这个列不能是 CHAR/ NCHAR/VARCHAR/VARNCHAR/ LONGVARCHAR/LONGNVARCHAR 类型
--incremental (mode)	指定 sqoop 如何确定哪些行是新的，包括了 append 和 lastmodified 两个选项
--last-value (value)	指定了从上次导入后的检查列的最大值

当一个表上的数据持续增加时，我们可以使用 Sqoop 的 append 模式来把新增的数据放到 HDFS 上。假定这个表的 ID 列是持续增加的，那么，你可以指定 check-column 为 ID。通过 last-value 参数，我们可以导入所有大于这个值的行。lastmodified 模式适合表数据被更新了，更新后的时间戳列上有最新的时间。 下面是一个增量导入的例子，在已经导入了前 100000 行之后，再导入新增的行数：

```
  sqoop import --connect jdbc:mysql://db.foo.com/somedb --table sometable -
-where "id > 100000" --target-dir /incremental_dataset --append
```

7.3.3　将数据从 Oracle 导入 Hive

虽然 Sqoop 的主要功能是将数据库的数据导入到 HDFS 上，但是它也支持将数据导入到 Hive 上。它会生成 CREATE TABLE 语句，并在 Hive 上执行这个语句。下面是一个从 Oracle 导入数据到 Hive 的例子：

```
    sqoop import --connect jdbc:oracle:thin:@192.168.1.191:1521:HTBASE --
table EMPLOYEES --hive-import
```

下面的 create-hive-table 工具根据 INPCASE.INP_DIAG 表的定义，在 Hive 的 metastore 中定义了表 INP_DIAG1：

```
    sqoop create-hive-table --connect
jdbc:oracle:thin:@192.168.1.191:1521:HTBASE --username SYSTEM --password
admin --table INPCASE.INP_DIAG --hive-table INP_DIAG1
```

7.3.4 将数据从 Oracle 导入 HBase

通过指定“--hbase-table”参数，Sqoop 支持把数据导入到 HBase 数据库的表上。源表上的每行数据都被转化为在 HBase 输出表上的 put 操作。默认情况下，Sqoop 使用 split-by 列作为行键列。如果没有指定这个参数，Sqoop 将使用主键列（如果有的话）。通过“—hbase-row-key”参数也可以指定行键列。下面是一个将数据从 Oracle 导入到 HBase 的例子：

```
    sqoop import --connect jdbc:oracle:thin:@192.168.1.191:1521:HTBASE --
username SYSTEM --password admin --m 1 --table INPCASE.PROGRESS_NOTE --
columns
pn_sn,ipid,pid,pn_date_time,pn_type_code,pn_type_desc,higher_id,higher_na
me,doctor_signature,submit_status,submit_time,submit_user_id,submit_user_
name,dept_code,dept_name,ward_code,ward_name,md5_content,create_time,crea
tor,modify_time,modifier,save_count,question_content,tpl_id,tpl_version,c
onfirm_status,confirm_time,confirm_user_id,confirm_user_name,case_code,ca
se_type,doctor_signature_id,creator_name,operation_code,operation_name,xg
sj,help_doctor_codes,inout_flag,relation_flow,first_submit_time,first_sub
mit_user_id,first_submit_user_name --hbase-create-table --hbase-table
INPCASE.PROGRESS_NOTE --hbase-row-key pn_sn --column-family info --split-
by ROWID
```

下面是一个 XML 类型的例子：

```
    sqoop import --connect jdbc:oracle:thin:@192.168.1.191:1521:HTBASE --
username oyt --password 123456 --m 1 --table ADMISSION_EVAL_REC --columns
rec_sn,ipid,pid,create_time,creator,modify_time,modifier,rec_cfg_sn,tpl_i
d,tpl_name,rec_date,recorder_id,recorder_name,xml_cont,dept_code,dept_nam
e,ward_code,ward_name,md5_content,question_content  --map-column-java
xml_cont=String  --hbase-create-table --hbase-table ADMISSION_EVAL_REC --
hbase-row-key rec_sn --column-family info --split-by ROWID
```

7.3.5　导入所有表

Sqoop 支持将数据库里的所有表导入到 HDFS 中，每个表在 HDFS 中都对应一个独立的目录。比如：

```
    sqoop  import-all-tables  -connect  jdbc:mysql://localhost:3306/test  -
hive-import
    sqoop import-all-tables --connect jdbc:oracle:thin:@192.168.1.191:1521:
HTBASE --username yzh --password 123456 --m 1
```

7.3.6　从 HDFS 导出数据

Sqoop 的 export 工具可以从 HDFS 导出数据到关系型数据库。目标的数据表必须在数据库中已经存在。Sqoop 的导出功能有三种模式：INSERT 模式（默认模式，Sqoop 创建 INSERT 语句）、UPDATE 模式（创建 UPDATE 语句）和 CALL 模式（为每个记录调用一次存储过程）。比如：

```
    sqoop export --connect jdbc:mysql://db.example.com/foo --table bar  --
export-dir /results/bar_data
```

上面的命令将/results/bar_data 下的文件中的数据导入到 foo 数据库的 bar 表中。-m 参数指定了配置多少个 mapper 来读取 HDFS 上文件块。在导出时，每个并行的 mapper 进程各自建立一个单独的数据库连接。每个语句将会插入 100 条记录，当完成 100 条语句也就是插入 10000 条记录，将会提交当前事务。

7.3.7　数据验证

Sqoop 提供了数据导出导入的验证功能，能够比较数据源与目标之间的行数。比如：

```
    sqoop import --connect jdbc:mysql://db.foo.com/corp  --table EMPLOYEES
-validate

    sqoop export --connect jdbc:mysql://db.example.com/foo --table bar  --
export-dir /results/bar_data -validate
```

对于导入功能，数据验证功能目前只能验证从一个表到 HDFS 的数据拷贝。

7.3.8　其他 Sqoop 功能

Sqoop 还提供了其他功能。比如：sqoop-merge 可以合并两个数据集的数据。sqoop-eval 让我

们只是执行对数据库的查询，并在控制台上打印出结果来。这样的话，就可以让我们验证这些查询语句：

```
sqoop eval --connect jdbc:mysql://db.example.com/corp --query "SELECT
* FROM employees LIMIT 10"
```

sqoop list-databases 可列出服务器上的数据库信息。比如：

```
sqoop list-databases --connect jdbc:mysql://database.example.com/
```

sqoop list-tables 可列出数据库上的表信息。比如：

```
sqoop list-tables --connect jdbc:mysql://database.example.com/corp
```

sqoop help 列出所有的工具信息。比如：

```
$ bin/sqoop help import
usage: sqoop import [GENERIC-ARGS] [TOOL-ARGS]

Common arguments:
   --connect <jdbc-uri>     Specify JDBC connect string
   --connection-manager <class-name>     Specify connection manager
class to use
   --driver <class-name>    Manually specify JDBC driver class to use
   --hadoop-mapred-home <dir>  Override $HADOOP_MAPRED_HOME
   --help                   Print usage instructions
   --password-file          Set path for file containing authentication
password
   -P                       Read password from console
   --password <password>    Set authentication password
   --username <username>    Set authentication username
   --verbose                Print more information while working
   --hadoop-home <dir>      Deprecated. Override $HADOOP_HOME

Import control arguments:
   --as-avrodatafile        Imports data to Avro Data Files
   --as-sequencefile        Imports data to SequenceFiles
   --as-textfile            Imports data as plain text (default)
   --as-parquetfile         Imports data to Parquet Data Files
...
```

sqoop-version 可以打印出 Sqoop 的版本信息。

7.4 Storm

Hadoop 作为一个擅长批量离线处理的框架，不适合海量数据的实时处理，而流处理框架的出现恰恰能满足这一点。在数据流模型中，需要处理的输入数据（全部或部分）并不存储在可随机访问的磁盘或内存中，它们以一个或多个“连续数据流”的形式到达（比如：视频流）。数据流模型的特点在于：

- 流中的数据元素在线到达，需要实时处理。
- 系统无法控制将要处理的、新到达的数据元素的顺序，无论这些数据元素是在一个数据流中还是跨多个数据流。
- 数据流的潜在大小也许是无穷无尽的。
- 一旦数据流中的某个元素经过处理，要么丢弃，要么被归档存储。

Storm 就是一套专门用于事件流处理的分布式计算框架，由 Twitter 贡献，于 2014 年 9 月正式成为 Apache 旗下的顶级项目之一。Storm 大大简化了面向庞大规模数据流的处理机制，从而在实时处理领域扮演着 Hadoop 之于批量处理领域的重要角色。Storm 支持容错和水平扩展。

Storm 是由 Clojure 和 Java 编写而成，设计目标在于支持将“流”（spout 或者 Spout，即输入流模块）与“栓”（bolt 或者 Bolt，即处理与输出模块）结合在一起并构成一套有向无环图（简称 DAG）拓扑结构。Storm 的拓扑结构运行在集群之上，而 Storm 调度程序则根据具体拓扑（Topology）配置，将处理任务分发给集群当中的各个工作节点。Storm 保证每个消息至少能够得到一次完成的处理。任务失败时，它会负责从消息源重试消息。Storm 的应用场景主要为以下三类：

- 信息流处理（streaming processing）：Storm 可用来实时处理新数据和更新数据库，兼顾容错性和扩展性。不像其他的流处理系统，Storm 不需要中间队列；
- 持续计算（Continuous computation）：Storm 可进行持续查询并把结果即时反馈给客户端；
- 分布式远程程序调用（Distributed RPC）：当 Storm 收到一条调用信息后，会对查询进行计算，并返回查询结果。

Storm 的关注重点放在了实时、以流为基础的处理机制上，因此其拓扑结构默认永远运行或者说直到手动中止。一旦拓扑流程启动，挟带着数据的流就会不断涌入系统，并将数据交付给栓（而数据仍将在各栓之间循流程继续传递），这正是整个计算任务的主要实现方式。随着处理流程的推进，一个或者多个栓会把数据写入至数据库或者文件系统当中，并向另一套外部系统发出消息或者将处理获得的计算结果提供给用户。

7.4.1 Storm 基本概念

Storm 是一个分布式的、可靠的、容错的数据流处理系统。Storm 主要包含有几个术语：spout、bolt、topology、streams、task、worker。在 Storm 中，我们首先要设计一个用于实时计算的图状结构，一般称之为拓扑（Topology）。这个拓扑将会被提交给集群，由集群中的主控节点（master node）分发代码，将任务分配给工作节点（worker node）执行。在 Java 代码中，我们可以通过 TopologyBuilder 类来构建拓扑。

一个拓扑中包括 spout 和 bolt 两种组件（角色），每个组件负责处理一项简单特定的任务。其中 spout 发送消息，负责将数据流以 tuple（元组）的形式发送出去；而 bolt 则负责转换这些数据流，在 bolt 中可以完成计算、过滤等操作，bolt 自身也可以将数据发送给其他 bolt。因此，Storm 集群的输入流由 spout 组件管理，spout 把数据传递给 bolt，bolt 要么把数据保存到某种存储器，要么把数据传递给其他的 bolt。你可以想象一下，一个 Storm 集群就是在一连串的 bolt 之间转换 spout 传过来的数据。下面图 7-11 所示是一个 Topology 的结构示意图。

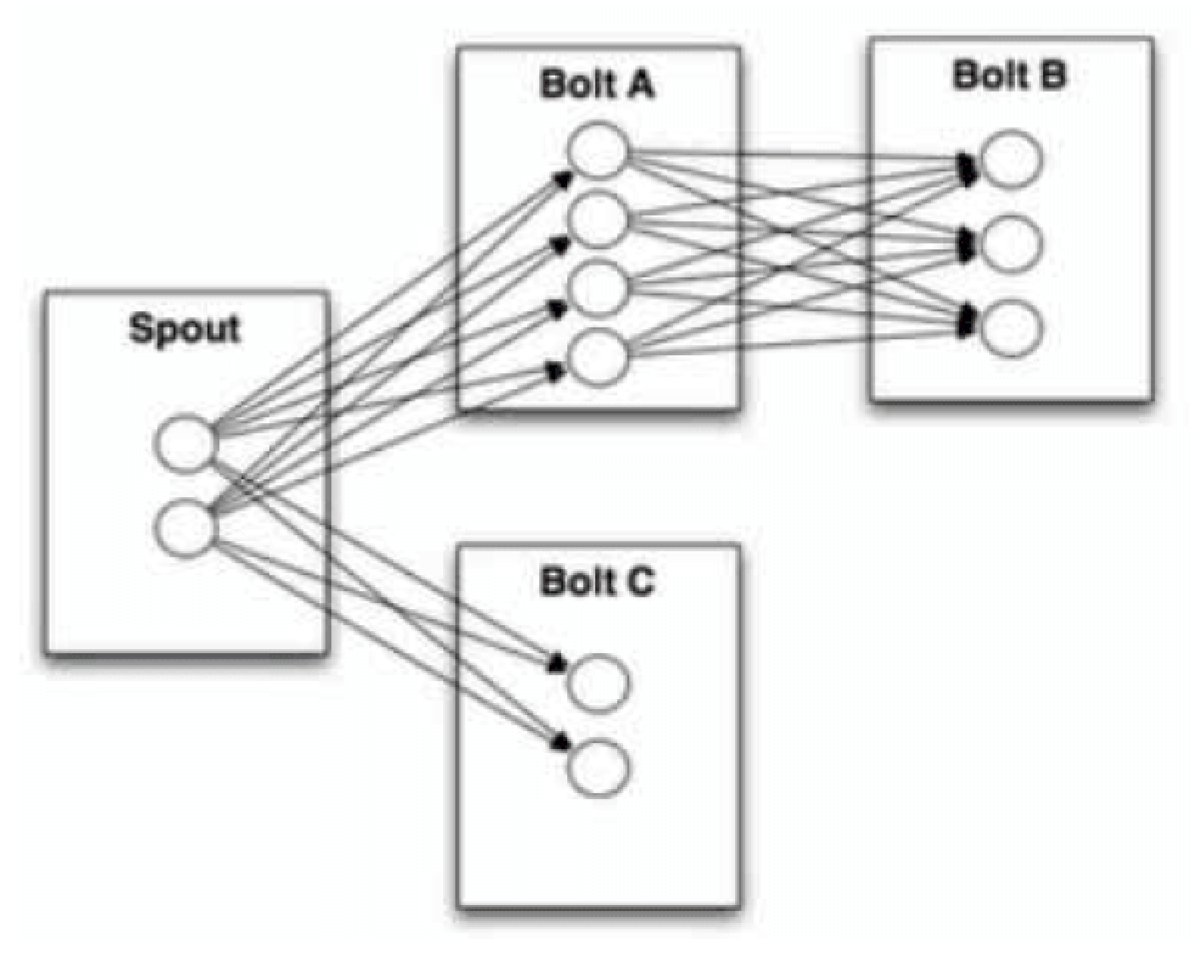


图 7-11　拓扑示意图

如图 7-11 所示，Spout 作为 Storm 中的消息源，用于为 Topology 生产消息（数据），一般是从外部数据源（如 Message Queue、RDBMS、NoSQL、日志文件）不间断地读取数据并发送 Topology 消息（tuple 元组）。而 Bolt 作为 Storm 中的消息处理者，用于为 Topology 进行消息的处理，Bolt 可以执行过滤、聚合、查询数据库等操作，而且可以一级一级地进行处理。最终，Topology 会被提交到 Storm 集群中运行；也可以通过命令停止 Topology 的运行，将 Topology 占用的计算资源归还给 Storm 集群。

数据流（Stream）是 Storm 中对数据进行的抽象，它是 tuple 元组序列。在 Topology 中，Spout 是 Stream 的源头，负责为 Topology 从特定数据源发射 Stream；Bolt 可以接收任意多个 Stream 作为输入，然后进行数据的加工处理过程，如果需要，Bolt 还可以发射出新的 Stream 给下级 Bolt 进行处理。

Topology 中每一个计算组件（Spout 和 Bolt）都有一个并行执行度，在创建 Topology 时可以

指定，Storm 会在集群内分配对应并行度个数的 task 线程来同时执行这一组件。

一个 Spout 或 Bolt 都会有多个 task 线程来运行，那么如何在两个组件（Spout 和 Bolt）之间发送 tuple 元组呢？Storm 提供了若干种数据流分发（Stream Grouping）策略用来解决这一问题。在 Topology 定义时，需要为每个 Bolt 指定接收什么样的 Stream 作为其输入（注：Spout 并不接收 Stream，只会发射 Stream）。目前 Storm 中提供了以下几种 Stream Grouping 策略：

- Shuffle Grouping：随机数据流组，这是最常用的数据流组。它只有一个参数（数据源组件），并且数据源会向随机选择的 bolt 发送 tuple 元组，保证每个消费者收到近似数量的元组。
- Fields Grouping：域数据流组，它允许你基于元组的一个或多个域控制如何把元组发送给 bolt。它保证拥有相同域组合的值集发送给同一个 bolt。在下节的单词计数器的例子中，使用了这种域数据流分组，它只会把相同单词的元组发送给同一个 bolt 实例。
- All Grouping：全部数据流组，为每个接收数据的实例复制一份元组副本。这种分组方式用于向 bolt 发送信号。比如，你要刷新缓存，你可以向所有的 bolt 发送一个刷新缓存信号。
- Global Grouping：全局数据流组，把所有数据源创建的元组发送给单一目标实例。
- None Grouping：不分组。
- Direct Grouping：直接数据流组，这是一个特殊的数据流组，数据源可以用它决定哪个组件接收元组。比如，数据源将根据单词首字母决定由哪个 bolt 接收元组。

7.4.2　spout

我们以一个实例来解释 Storm 的各个组件。这个程序是创建一个简单的拓扑，用于计算单词数量。spout 的代码如下：

```
    public class WordReader implements IRichSpout {
        private SpoutOutputCollector collector;
        private FileReader fileReader;
        private boolean completed = false;
        private TopologyContext context;
        public boolean isDistributed() {return false;}
        public void ack(Object msgId) {System.out.println("OK:"+msgId); }
        public void close() {}
        public              void              fail(Object              msgId)
{ System.out.println("FAIL:"+msgId);}

        //创建一个读文件对象，并维持一个 collector 对象
        //这是第一个被调用的 spout 方法，它接收如下参数：配置对象，在定义 topology
        // 对 象 时 创 建 ； TopologyContext 对 象 ， 包 含 所 有 拓 扑 数 据 ；
SpoutOutputCollector
```

```
        //对象，它能让我们发布交给 bolts 处理的数据。
        public void open(Map conf, TopologyContext context,
SpoutOutputCollector collector) {
            try {
                this.context = context;
                this.fileReader = new
FileReader(conf.get("wordsFile").toString());//读取文件
            } catch (FileNotFoundException e) {
                throw new RuntimeException("Error reading file
["+conf.get("wordFile")+"]");
            }
            this.collector = collector;
        }

        //这个方法是读取文件并逐行发布数据，通过这个方法向 bolts 发布待处理的数据。
        //这个方法会不断地被调用，直到整个文件都读完了。
        public void nextTuple() {
        //nextTuple()会被 ack()和 fail()周期性的调用。没有任务时它必须释放对线程的控
        //制，其他方法才有机会得以执行。因此 nextTuple 的第一行就要检查是否已处理
        //完成。如果完成，则文件中的每一行都已被读出并分发了。
            if(completed){
                try {
                    Thread.sleep(1000);// 如果完成，会休眠一毫秒，以降低处理器负载
                } catch (InterruptedException e) {
                    //什么也不做
                }
                return;
            }
            String str;
            //创建 reader
            BufferedReader reader = new BufferedReader(fileReader);
                try{
                    //读所有文本行
                    while((str = reader.readLine()) != null){
                        //按行发布一个新值
                        this.collector.emit(new Values(str),str);
                    }
                }catch(Exception e){
                    throw new RuntimeException("Error reading tuple",e);
```

```
            }finally{
              completed = true;
            }
        }

        //声明输入域"word"
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
              declarer.declare(new Fields("line"));
        }
  }
```

spout 是输入流模块，读取原始数据，为 bolt 提供数据。spout 最终会发送一个流（Stream），就是文件中的一行。上面代码中的注解详细解释了每个方法的作用和用法。

7.4.3　bolt

现在我们有了一个 spout，用来按行读取文件并按照每行发布一个元组。我们还要创建两个 bolt，第一个 bolt 用来标准化单词，第二个 bolt 为单词计数。bolt 最重要的方法是 void execute(Tuple input)，每次接收到元组时都会被调用一次，还会再发布若干个元组。

第一个 bolt，WordNormalizer，负责接收并标准化每行文本。它把文本行切分成单词，大写转化成小写，去掉头尾空白符。代码如下：

```
   public class WordNormalizer implements IRichBolt{
      private OutputCollector collector;
      public void cleanup(){}

      //处理传入的元组:bolt 从单词文件接收到文本行，并标准化它。
      //文本行会全部转化成小写，并切分它，从中得到所有单词。
      public void execute(Tuple input){
          String sentence = input.getString(0); //从元组读取值
          String[] words = sentence.split(" ");
          for(String word : words){
              word = word.trim();
              if(!word.isEmpty()){
                  word=word.toLowerCase();
                  //发布这个单词
                  List a = new ArrayList();
                  a.add(input);
                  collector.emit(a,new Values(word));
              }
```

```
            }
            //每次都调用 collector 对象的 ack()方法确认已成功处理了一个元组
            collector.ack(input);
        }
        public  void  prepare(Map  stormConf,  TopologyContext  context,
OutputCollector collector) {
               this.collector=collector;
        }

        //声明 bolt 只会发布一个名为“word”的域
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
            declarer.declare(new Fields("word"));
        }
    }
```

上面这个代码是在一次 execute 调用中发布多个元组。如果这个方法在一次调用中接收到句子“This is Samuel Yang in San Jose”，它将会发布 7 个元组。

下一个 bolt，WordCounter，负责为单词计数。当拓扑结束时（cleanup()方法被调用时），它将显示每个单词的数量。这个例子的 bolt 什么也没发布，它把数据保存在 map 里，但是在真实的场景中可以把数据保存到数据库。

```
    public class WordCounter implements IRichBolt{
        Integer id;
        String name;
        Map<String,Integer> counters;
        private OutputCollector collector;

        //拓扑结束时（集群关闭的时候），显示单词数量
        //通常情况下，当拓扑关闭时，应当关闭活动的连接和其他资源
        public void cleanup(){
            System.out.println("-- 单词数 【"+name+"-"+id+"】 --");
            for(Map.Entry<String,Integer> entry : counters.entrySet()){
                System.out.println(entry.getKey()+": "+entry.getValue());
            }
        }

        //使用一个 map 收集单词并计数
        public void execute(Tuple input) {
            String str=input.getString(0);
```

```
        //如果单词尚不存在于 map，我们就创建一个，如果已在，我们就为它加1
        if(!counters.containsKey(str)){
            conters.put(str,1);
        }else{
            Integer c = counters.get(str) + 1;
            counters.put(str,c);
        }
        //对元组作为应答
        collector.ack(input);
    }

    //初始化
    public  void  prepare(Map  stormConf,  TopologyContext  context,
OutputCollector collector){
        this.counters = new HashMap<String, Integer>();
        this.collector = collector;
        this.name = context.getThisComponentId();
        this.id = context.getThisTaskId();
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {}
}
```

从上面的例子可以看出，Bolt 是这样一种组件，它把元组作为输入，然后产生新的元组作为输出。Bolt 拥有如下方法：

```
//为 bolt 声明输出模式
declareOutputFields(OutputFieldsDeclarer declarer)
//仅在 bolt 开始处理元组之前调用
prepare(java.util.Map stormConf, TopologyContext context,
OutputCollector collector)
//处理输入的单个元组
execute(Tuple input)
//在 bolt 即将关闭时调用
cleanup()
```

7.4.4　拓扑

Topology 是拓扑结构，为 Storm 的一个任务单元。下面我们在主类中创建这个拓扑和一个

本地集群对象。我们要用一个 spout 读取文本，第一个 bolt 用来标准化单词，第二个 bolt 为单词计数。这个拓扑决定 Storm 如何安排各节点，以及它们交换数据的方式。为了便于在本地测试和调试，LocalCluster 可以通过 Config 对象，尝试不同的集群配置。

```
    public class TopologyMain {
       public static void main(String[] args) throws InterruptedException
{
       //创建一个拓扑
          TopologyBuilder builder = new TopologyBuilder());
          builder.setSpout("word-reader", new WordReader());
          //在 spout 和 bolts 之间通过 shuffleGrouping 方法连接
          //这种分组方式决定了 Storm 会以随机分配方式从源节点向目标节点发送消息
          builder.setBolt("word-normalizer",
    new WordNormalizer()).shuffleGrouping("word-reader");
          builder.setBolt("word-counter",
    new WordCounter(),2).fieldsGrouping("word-normalizer", new
Fields("word"));
          //创建一个包含拓扑配置的 Config 对象，它会在运行时与集群配置合并
           //并通过 prepare 方法发送给所有节点
          Config conf = new Config();
          conf.put("wordsFile", args[0]); //由 spout 读取的文件的文件名，赋值给
wordFile 属性
          //在开发阶段，可设置 debug 属性为 true，Strom 会打印节点间交换的所有
          //消息，以及其他有助于理解拓扑运行方式的调试数据
          conf.setDebug(false);

          //运行拓扑
          conf.put(Config.TOPOLOGY_MAX_SPOUT_PENDING, 1);
          LocalCluster cluster = new LocalCluster();
           cluster.submitTopology("Getting-Started-Topologie", conf,
builder.createTopology();
           Thread.sleep(1000);
           cluster.shutdown();
       }
    }
```

在生产环境中，拓扑会持续运行。对于上面这个例子而言，你只要运行它几秒钟就能看到结果。最后几行代码是调用 createTopology 和 submitTopology，运行拓扑，休眠一秒钟（拓扑在另外的线程运行），然后关闭集群。

在上面这个例子中，每类节点只有一个实例。但是如果你有一个非常大的文件呢？你能够很轻松地改变系统中的节点数量实现并行工作。这个时候，你就要创建两个 WordCounter 实例：

```
    builder.setBolt("word-counter", new
WordCounter(),2).shuffleGrouping("word-normalizer");
```

每个实例都会运行在单独的机器上。当你调用 shuffleGrouping 时，就决定了 Storm 会以随机分配的方式向你的 bolt 实例发送消息。在上面这个例子中，理想的做法是相同的单词发送给同一个 WordCounter 实例。只要把 shuffleGrouping(“word-normalizer”) 换成 fieldsGrouping(“word-normalizer”, new Fields(“word”))就能达到目的。

7.4.5　Storm 总结

在 Storm 集群中，有两类节点：主节点 master node 和工作节点 worker nodes。主节点运行着一个叫做 Nimbus 的守护进程。这个守护进程负责在集群中分发代码，为工作节点分配任务，并监控故障。Supervisor 守护进程作为拓扑的一部分运行在工作节点上。一个 Storm 拓扑结构在不同的机器上运行着众多的工作节点。

Storm 生态系统的一大优势在于其拥有丰富的流类型组合，足够从任何类型的来源处获取数据。Storm 适配器的存在使其能够轻松与 HDFS 文件系统进行集成，可以与 Hadoop 实现互操作。Storm 的另一大优势在于它对多语言编程方式的支持能力。尽管 Storm 本身基于 Clojure 且运行在 JVM 之上，其输入流与栓仍然能够通过几乎所有语言进行编写。

总之，Storm 是一套极具扩展能力、快速且具备容错能力的开源分布式计算系统，其高度专注于流处理领域。Storm 在事件处理与增量计算方面表现突出，能够以实时方式根据不断变化的参数对数据流进行处理。尽管 Storm 同时提供原语以实现通用性分布 RPC，并在理论上能够被用于任何分布式计算任务的组成部分，但其最为根本的优势仍然表现在事件流处理方面。

Storm 有着非常快的处理速度，单节点可以达到每秒百万个元组，此外，它还具有高扩展、容错、保证数据处理等特性。实时数据处理的应用场景很广泛，例如商品推荐、广告投放等，它能根据当前情景上下文（用户偏好、地理位置、已发生的查询和点击等）来估计用户点击的可能性并实时做出调整。

7.5　Splunk

Splunk 既是 NASDAQ 上市公司名称，也是产品名称。使用 Splunk 可以收集、索引和分析所有应用程序、服务器和设备上生成的实时数据（主要是日志数据），并完成所谓的“Operational Intelligence（智能化运营）”。Splunk 能生成各类报告、图表和仪表板，以便监视安全事件和攻击，监视应用程序 SLA 和其他关键性能指标，监视用户行为、机器行为、安全威

胁、欺诈活动等。使用 Splunk 可监视基础结构，避免服务性能降低或中断，关联并分析跨越多个系统的复杂事件。需要提醒读者注意的是，Splunk 不是一个开源的系统。

Splunk 主要是给 IT 部门使用的一个托管的日志文件管理工具，它的主要功能包括：

- 日志聚合功能。
- 搜索功能。
- 提取功能。
- 对结果进行分组、联合、拆分和格式化。
- 可视化功能。
- 分析（统计、监控、报警、安全事件审计、DDoS 攻击监控等）。

Splunk 是一个分布式的机器数据平台，主要有三个角色：

- Search Head，负责数据的搜索和处理，提供搜索时的信息抽取。
- Indexer，负责数据的存储和索引
- Forwarder，负责数据的收集，清洗，变形，并发送给 Indexer。

第 8 章

大数据管理平台

大数据分析指从海量的原始数据抽取出有价值的信息，是将大数据转换成信息的过程。主要对所输入的各种形式的海量数据进行加工分析，其过程包含对数据的收集、存储、加工、分类、归并、计算、排序、转换、检索和可视化的全过程。大数据分析离不开数据质量和数据管理，高质量的数据和有效的数据管理，才能保证分析结果的真实和有价值。还有，非结构化数据的多元化给大数据分析带来新的挑战，我们需要一套工具系统综合管理结构化和非结构化数据。

大数据管理是指大数据的收集整理、组织、存储、维护、检索、传送等操作，是大数据分析的基础环节，也是所有大数据分析过程中必有的共同部分。大数据分析因业务的不同而不同，需要根据业务的需要来编写应用程序加以解决。

大数据管理比较复杂，由于可利用的数据呈爆炸性增长，且数据的种类繁杂，从数据管理角度而言，不仅要使用数据，而且要有效地管理数据。这就需要一个通用的、使用方便且高效的管理软件，把数据有效地管理起来。数据分析与数据管理是相互联系的，数据管理技术的优劣将对数据分析的效率产生直接影响。

8.1 大数据建设总体架构

目前，政府的信息化存在基础设施和系统建设分散，应用“烟囱”和数据“孤岛”林立，业务协同和信息资源开发利用水平低，综合支撑和公众服务能力弱等突出问题，难以适应和满足新时期工作需求。为了充分运用大数据、云计算等现代信息技术手段，全面提高政府综合决策、监管治理和公共服务水平，加快转变管理方式和工作方式，政府正在力推大数据建设。

大数据建设需要加强顶层设计和统筹协调，完善标准体系，统一基础设施建设，推动信息资源整合互联和数据开放共享，促进业务协同，推进大数据创新应用，保障数据安全。大数据建设需要以大数据管理平台为核心，统筹整合内外数据资源，边整合边应用。

如图 8-1 所示，大数据总体架构为“一个机制、两套体系、三个平台”。一个机制即大数据管理工作机制，两套体系即组织保障标准规范体系、统一运维和信息安全体系，三个平台即大数据云平台、大数据管理平台和大数据应用平台。



图 8-1　大数据建设框架

这一个机制、两套体系、三个平台具体来说就是：

- 一个机制：大数据管理工作机制包括数据共享开放、业务协同等工作机制，以及大数据科学决策、精准监管和公共服务等创新应用机制，促进大数据形成和应用。大数据管理工作机制健全了大数据标准规范体系，保障了数据准确性、一致性和真实性，强化了安全防护，保障了信息安全。
- 两套体系：组织保障标准规范体系为大数据建设提供组织机构、人才资金及标准规范等体制保障；统一运维信息安全体系为大数据系统提供稳定运行与安全可靠等技术保障。
- 三个平台：大数据平台分为基础设施层、数据资源层和业务应用层。其中，云平台是集约化建设的 IT 基础设施层，实施网络资源、计算资源、存储资源、安全资源的集约建设、集中管理、整体运维，为大数据处理和应用提供统一基础支撑服务；大数据管理平台是数据资源层，为大数据应用提供统一数据采集、分析和处理等支撑服务；大数据应用平台是业务应用层，为大数据在各领域的应用提供综合服务。

通过大数据建设和应用，实现综合决策科学化。将大数据作为支撑管理科学决策的重要手段，为政策计划提供信息支持，实现“用数据决策”。利用大数据支撑实现管理精准化，实现“用数据管理”，提高管理的主动性、准确性和有效性。实现政府公共服务便民化。运用大数据创新政府服务理念和服务方式，实现“用数据服务”。利用大数据支撑信息公开、网上一体化办事和综合信息服务，建立公平普惠、便捷高效的公共服务体系，提高公共服务共建能力和共享水平，发挥数据资源对人民群众生产、生活和经济社会活动的服务作用。

8.2 大数据管理平台的必要性

大数据建设需要大数据管理平台，这是因为：

（1）信息孤岛期待数据整合

烟囱式的系统建设导致政府和企业建设了多个相互独立的系统，这些系统之间由于是不同厂商开发，而且分属不同时间段开发，相互之间没有统一的数据交互标准，导致许多数据没有统一

规划，无法有效交互，导致同一数据分散管理，含义不同。大数据管理平台可以帮助政府和企业建设系统交互标准，实现同一数据的统一管理入口，并通过数据实时交互将数据发送给所有需要数据的一方，数据需求方可实现本地存储，提高数据查询检索效率。

（2）渐进式的大数据应用系统建设需要大数据管理平台

没有人能够在初期就完全规划所有大数据应用系统的建设，注定了应用系统建设是一个渐进式的过程，每新建一个应用系统无可避免地需要从已有系统中获取数据才能实现有效集成。而传统方式是每个新系统的上线都需要所有系统的开发商为其定制化开发接口，大大地增加了系统开发复杂度、提高了项目费用、延后了项目工期。有了大数据管理平台后，当新系统上线时不再需要已有系统提供数据接口，而是在数据平台上通过参数的设置简单实现从已有系统上实时获取数据。

（3）更大深度地挖掘数据的价值需要大数据管理平台

大数据管理平台为政府和企业建设集中统一的大数据平台，从各个业务系统中实时获取数据进行加工处理，发现数据中蕴含的巨大价值。

8.3 大数据管理平台的功能

大数据的核心是利用技术，把“数据”这个资源充分利用起来，让其发挥其应有的作用，为企业的发展、政府的服务提供价值。要想实现上述大数据的价值，首先要做的就是如何规划、整理、处理这些数据。大数据管理平台自然而然成为基础和核心。

大数据管理平台是数据资源传输交换、存储管理和分析处理的平台，为大数据应用提供统一的数据层。它主要实现数据传输交换、管理监控、共享开放、分析挖掘等基本功能，支撑分布式计算、流式数据处理、大数据关联分析、趋势分析、空间分析，支撑大数据应用产品的研发。大数据管理平台主要实现的三大功能分小节说明如下。

8.3.1 推进数据资源全面整合共享

提升数据资源获取能力。加强数据资源规划，明确数据资源采集目标，建立数据采集目录，避免重复采集，逐步实现“一次采集，多次应用”。利用物联网、移动互联网等新技术，拓宽数据获取渠道，创新数据采集方式。

加强数据资源整合。破除数据孤岛，建立信息资源目录体系，实现系统内数据资源整合集中和动态更新，建设各类基础数据库。通过政府数据统一共享交换平台接入人口基础信息库、法人单位资源库、自然资源和空间地理基础库等其他国家基础数据资源。拓展吸纳外部合作单位和互联网关联数据，形成信息资源中心，实现数据互联互通。

推动数据资源共享服务。明确各部门数据共享的范围边界和使用方式，厘清各部门数据管理

及共享的义务和权力，制定数据资源共享管理办法，编制数据资源共享目录。提供灵活多样的数据检索服务，形成向平台直接获取为主、部门间数据交换获取为辅的数据共享机制，提高数据共享的管理和服务水平。

推进数据开放。建立数据开放目录，推动政府向社会开放部分数据，提高数据开放的规范性和权威性。

8.3.2 增强数据管理水平

按照行业数据标准规范体系，建立平台级数据和业务标准模型，保障数据准确性、一致性、真实性和权威性。建立统一的访问机制，合理规范业务数据的使用方式与范围。

建立集中统一的信息安全管理平台，明确数据采集、传输、存储、使用、开放等各环节的信息安全范围边界、责任主体和使用权限。落实信息安全等级保护等国家信息安全制度。增强数据资源和应用系统等的安全保障能力。

实施数据管理制度，明确各部门数据的有效期和时效性，加强数据版本化管理，增强数据加密能力，提供数据多维归类功能，强化原始数据的不可更改性，提供数据行为审计。

提供数据处理流程引擎，为各类数据的处理提供可配置的标准处理流程。

8.3.3 支撑创新大数据分析

为数据资源开发与应用提供统一的访问服务，创新大数据分析与应用，支撑精细化分析和实时可视化表达，增强趋势分析和预警能力，为决策和管理提供数据支持，提高管理决策预见性、针对性和时效性。

提供自动多维归类功能，加强各类数据的关联分析，包括与外部数据资源（如：合作单位数据、互联网数据、购买数据）融合利用和信息服务，提高业务处理能力。利用跨部门的数据资源，支撑定量化、可视化评估实施成效。

快速搜集和处理涉及业务风险、突发事件、社会舆论等海量数据，综合利用各部门的数据，开展大数据统计分析，构建大数据分析模型，建设大数据应用，提升决策等能力。

8.4 数据管理平台（DMP）

DMP，全名为 Data-Management Platform，即数据管理平台，是利用大数据技术从海量杂乱的数据中抽取出有价值信息的重要基础设施。DMP 是专门针对广告投放的数据管理平台，几乎所有的大型广告公司都将它用于与 DSP（Demand-Side Platform）配合来优化广告投放效果。

随着数据时代的到来，DMP 开始从早期广告服务平台逐步演变成为企业客户营销的核心引擎。DMP 更多被定义为能统一抽取公司各业务离散的数据并作出科学分析来支撑决策的技术性

平台。DMP 能够整合集成各类基础业务数据，如客户数据、会员数据、ERP 数据、DEM 数据、用户在网页和 APP 上的访问数据等，并利用模型算法来帮助行业客户从海量数据中挖掘到有价值的商业信息，给予产品推广和营销工作支持。DMP 的这种数据采集、分析处理、应用反馈的回路也是周而复始，而且有机会使系统变得越来越敏锐和智慧。

DMP 一定要在使用中才能产生价值，那么除了 RTB 广告，DMP 主要有以下两种变现途径：

- 数据报告变现：这种模式不涉及个体用户的隐私，因此相对比较成熟，比如通过对于人流量的分析可以广泛地用于交通流量预测、旅游流量预测、商家选址等等，可以为政府、商家提供相关的咨询服务。
- 数据服务变现：通过数据运算将企业的第一方数据和数据管理平台内的数据融合，不断地挖掘其中的价值，深度洞察用户和寻找潜在客户，将企业沉睡数据的价值发挥最大化。数据洞察和基于数据的分析服务，已经成为 DMP 越来越重要的应用方向，而且成为评判 DMP 优劣的最直观应用。

从数据的专业性来说，DMP 不仅要包含大数据，还要包括传统业务数据等多种数据载体，除了 PC 端还有移动端，除了企业的第一方数据还要包括企业外部的数据来源（如社交媒体）等非结构化数据内容，掌控用户在企业之外的数据存在状态，识别哪些数据可以用于后续处理分析。

高价值数据体现在两个方面，一个是数据的连续性，同一用户在不同站点上的行为数据能够被识别并连接在一起；另一个是多种终端的数据的连续性，目前主要还是 PC 端和手机端。对于第一个数据，由于来自于自身业务开展过程中的数据积累，其数据质量往往比较高，如果数据量够大，数据维度也够丰富，完全可以对外提供 DMP 数据服务。国内有一些拥有庞大数据的机构，目前还是更多地把自身的数据脱敏后售卖给第三方 DMP 公司。无论哪种数据来源，都并非直接对接给需求方，而是采用统一化的方式将各方数据吸纳整合，再进行数据处理和融合，做标准化、结构化的细分。这样加工后再推向营销和分析环境中的数据才能更完整，系统性也更强，同时也不再是原始数据的形态。

无论是对外提供哪种 DMP 数据服务，其算法和模型都是建立在所拥有的数据之上，数据基础越是丰富、建模能力越强、对行业理解越深入，就越能准确地捕捉用户在互联网上的行为，建立更完善的标签分类体系，为用户更好地画像，建立适应行业需要的业务应用模型，大数据应用才能扎实地展开。好的 DMP 产品能解决企业的实际业务问题，优秀的数据管理平台不仅仅只是提供一个技术平台，更重要的是需要具备业务理解能力和分析建模能力，只有这样才能真正用好这些数据。

与上节的大数据管理平台的功能要求相比，DMP 只是完成了大数据管理平台的一部分功能。DMP 还不能算是大数据管理平台，它只是一个为企业营销服务的数据管理软件。而大数据管理平台给企业和政府提供了一个只关心数据的数据层。通过大数据管理平台，帮助企业整合它们所有的数据，并在一个平台上展现和汇聚所有数据。从前在 IT 系统上不存在数据层，一个企业的信息中心或 CIO 想要看自己的企业的所有数据，基本没有地方看（银行业通过数据仓库来看，有 T+1 的延时）。企业的统一数据平台给企业带来了很多好处，好处之一就是未来的所有新

应用就可以真正基于一个平台来开发了。虽然 Hadoop 本身提供了大数据管理平台的基础软件，但是我们要避免把 Hadoop 用成是“一个”系统软件，而忽视统一数据平台的重要性。以应用为导向的思路，很容易让客户采用“Hadoop+应用”的架构，对于一套套应用系统，形成了多个“Hadoop+应用”的烟囱式系统，就像当年一个企业部署多个不同 Oracle 应用的情形一样。

8.5 EasyDoop 案例分析

我们以业界知名的大数据管理平台 EasyDoop 为例，来阐述大数据管理平台的各个组件和其功能架构。从技术上讲，EasyDoop 大数据管理平台以 Hadoop 为基础，具备了大数据管理与处理分析的能力，提供了解决大数据应用需求的管理工具和集成、开发、展现组件，使企业可实现大规模结构化、非结构化数据的集中、一体化的分析处理需求。EasyDoop 基于 Hadoop、Hive、Storm、Spark、Flume、Sqoop、HBase、Phoneix、Kafka 等组件进行了商业化提升，涵盖大规模非结构化数据集成、存储、管理和分析计算。EasyDoop 集成 YARN，支持多种分布式计算框架（MapReduce、Spark 等），可轻松管理 PB 级数据。EasyDoop 提供了基于流式处理的实时分析，数据可在内存中处理，而不经过磁盘。EasyDoop 采用 DAG 计算模型，可以根据业务需要增减 bolt 组合计算流程。EasyDoop 支持大吞吐量，单集群的每个 bolt 可处理 10Gb/s。EasyDoop 基于 Hive 提供了非结构化数据的即时（ad-hoc）分析，包括查询和聚合等，支持文本、SequenceFile、RCFile 等文件格式，支持 GZIP 等压缩算法。

大数据系统首先要给数据建立模型，然后从多种数据源获取数据，数据的预处理（清洗、验证等），数据存储，数据处理和分析，最后呈现和可视化数据分析的结果。如图 8-2 所示，EasyDoop 大数据管理平台分为大数据建模平台、大数据交换和共享平台、大数据云平台和大数据服务平台，从而为用户构建了全域的统一数据资源层。

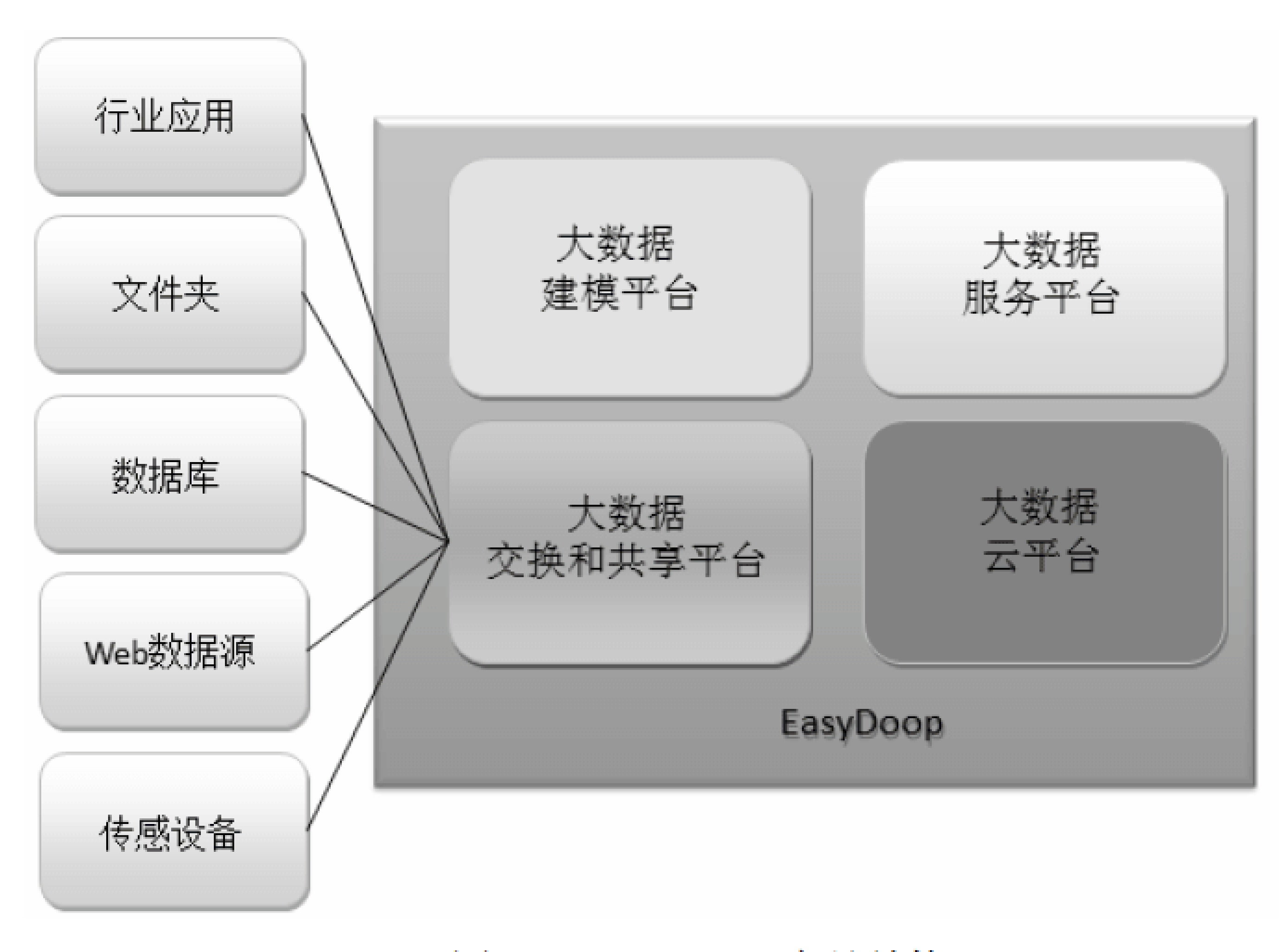


图 8-2　EasyDoop 产品结构

在大数据建模平台上，可快速建立各类数据模型和处理流程模型，配置各类数据的安全和归档规则。在大数据交换和共享平台上，可设置异构数据源的采集和映射规则，自动从庞大的内外数据源上采集数据到大数据云平台。大数据云平台是存储和管理大数据的平台，可横向扩展到数万台服务器。大数据服务平台提供数据服务功能和大数据分析工具。EasyDoop 提供了 API 和 Web 服务用于开发大数据创新应用。

8.5.1　大数据建模平台

建模平台包含了针对数据模型、业务流程模型、安全和访问控制模型、存储模型的创建、更新和删除等功能。

1. 数据模型建模

大数据系统需要从不同种类不同来源的数据进行采集，这些不同数据源获取的数据具有不同的格式，使用不同的协议。建模平台提供了统一的数据模型管理。统一的、标准的数据模型确保了数据是准确、可靠、值得信赖的，来龙去脉清楚，并且具有一致性，才能帮助分析决策。

EasyDoop 上的数据模型包含了非结构化数据和结构化数据，按照业务对象模型来建立数据模型。数据的版本控制、归档设置、多维自动归类、安全访问设置等都是以数据模型为中心开展起来的。系统支持数据属性的自定义校验规则，如数据的唯一性、关联性、完整性等。

数据模型可以按照业务类别归类到不同的系统空间上，也可以按照数据采集、数据存储、数据应用的逻辑顺序，归类到虚拟库下，比如：采集库、基础库、统计分析库。采集库是平台数据的入口，数据由大数据交换共享平台完成采集，首先进入到采集库中，经过清洗和校验后，作为基础数据进入到基础库中。基础库用于数据资源的集中存储和统一管理。统计分析库存放统计分析的结果数据，便于结果数据的可视化呈现。

EasyDoop 支持数据模型的动态调整。数据模型不是一成不变的，可以根据需求的变更而动态调整。在整个数据处理过程中，花在特征提取和选择上的时间比选择和实现算法的时间还要多。比如：在欺诈交易检测模型中，我们需要从许多可能的特征中进行选择，然后将特征转换成适用于机器学习算法的向量。

2. 业务流程建模

业务流程是数据处理的一系列步骤。这包括自动化处理流程与半自动化处理流程。EasyDoop 提供业务处理流程引擎，帮助政府和企业创建和管理自身的业务处理流程。EasyDoop 使得数据模型、业务处理流程和业务数据分离，从而提供了最大化的灵活性和便利性。

3. 存储建模

数据本身分为冷数据和热数据。存储介质有快有慢，有贵有便宜的。虽然大数据平台统一存储了各部门的数据，从数据安全性、成本效益和存储性能的角度考虑，有时又需要在存储上分开存放。EasyDoop 的存储建模就是为政府和企业提供虚拟设备的管理功能，为不同的数据模型提

供不同的存储。EasyDoop 的存储模型可以混合 Hadoop 分布式文件系统和传统文件系统，可以横向扩展多个存储介质，可以在平台级别上控制虚拟存储容器的读写设置。

4. 安全建模

所有数据都在同一个平台之后，数据的安全性显得更加重要。数据本身是一种资产，如果数据的敏感性和安全性没有保证，那么政府和企业不敢采用。所以，建模过程中也包含了对数据安全性的设置，这包括了数据的访问控制列表、加密和解密、统一的用户和权限管理等功能。不同部门的用户在统一的平台上登录、验证、访问授权数据。

8.5.2 大数据交换和共享平台

大数据交换和共享平台首先解决平台数据源的问题。根据数据采集原则，支持多种数据采集方式，获取不同格式的数据，集中入库到平台上，便于数据的统一管理和访问，便于开展共享数据服务。其次是打破业务系统条块分割和信息孤岛，解决各业务系统之间数据交互的问题。大数据交换和共享平台的特点为：

- 多样性：支持从各类主流数据库（Oracle、DB2、SQL Server、MySQL 等）、端口、设备、邮件系统、外部网站、URL、Web 服务、私有应用系统、文件系统上采集元数据和音频、视频、PDF、办公文件等非结构化资源，并保持同步更新。
- 高性能：每秒能采集几十万条数据，能横向扩展以支持更多更快的数据采集。
- 实时性：保证较低的延迟时间。在环保行业上， EasyDoop 的环保监测数据的采集达到了秒级，甚至是毫秒级别。
- 易用性：用户在无须了解 Hadoop 技术的情况下，可快速地进行开发和部署；管理界面简单易懂，用户通过简单的几步配置即可完成数据的采集。
- 可扩展性：EasyDoop 提供了数据源连接器的对外接口，用户基于这个接口既可开发自己企业应用系统的数据连接器，也能够访问企业外部的数据源。
- 可视化：支持采集作业与任务执行的可视化与分析，能够更好地查看进度和性能，保证数据采集始终处于有序管理的状态。
- 智能化：预先配置好关键的运行指标，可以直接查看采集任务是否健康执行。

大数据交换平台包括了以下的数据交换、采集管理和数据抓取等模块。

1. 数据交换

数据交换模块是指通过双方约定的接口进行数据的获取和发布。平台支持的数据交换接口包括：数据库接口、文件接口、API 接口、邮件服务器接口、设备接口等。

数据库接口有三种形式：数据库文件、数据库同步和数据库接口。对于数据库文件接口，数据厂家定期将全量或增量的数据库文件发布到指定的服务器（厂家 FTP 服务器或平台的采集前置机）中，采集程序获取文件，导入到数据库中；对于数据库同步接口，平台对数据厂家开放平

台访问权限，由数据厂家将数据定期同步至平台中；对于数据库接口，数据厂家对平台开放数据库访问权限，由平台主动获取厂家数据库中的数据。

对于文件接口，通过数据厂家提供的文件接口采集数据文件，对数据文件进行处理后存储到平台中。文件接口支持 FTP 接口、文件同步以及其他扩展接口。

对于 API 接口，通过数据厂家提供的 API 接口进行数据交换，包括 HTML 页面接口、Web 服务接口、Rest 接口等。

对于设备接口，数据厂家将数据发送到平台指定的端口上，平台获取后存储在平台上。或者，平台发送数据到数据厂家指定的端口上。

2. 采集管理

采集管理模块是对数据采集规则、数据源连接器、采集任务进行管理和监控。按照采集规则，对于采集的数据进行清洗转换，并存储到平台上。模块包括采集规则管理、连接器管理、采集服务管理和服务控制台四部分功能。

采集规则管理功能主要实现采集任务过程中相关参数的配置工作，如采集任务的类型、采集周期、采集时间频度、数据源位置信息、校验规则、数据的映射规则等参数，在采集过程中会按照事先配置好的这些参数要求完成数据的采集工作。

一个数据源有一个连接器。按照数据源的种类（如：Oracle 数据库、文件系统、监听端口等），连接器管理为每个数据源定义一个连接器。每个连接器实现与一种数据源的连接、数据采集、数据写入、数据清洗与转化等通用功能。

采集服务是采集规则、连接器和处理流程的集合，采集服务管理就是创建不同的采集服务。比如：5 个都在使用 Oracle 数据库的业务系统，则连接器只有一个（Oracle 连接器），但是采集服务和采集规则有 5 个，分别定义了从 5 个业务系统上采集不同的数据到平台上。采集服务管理包括任务的新增、修改、删除、查询以及停用等功能。

采集服务控制台主要实现检测所有采集任务从开始到结束的运行情况，通过此功能用户可以方便查看采集任务的状态情况，任务的状态可通过警示灯的颜色区分，成功状态为绿色，失败状态为红色，对于采集失败的任务，平台会及时通过邮件、短信、系统弹出框等方式通知和提醒管理员。通过失败信息，管理员就能了解任务失败的时间、失败的原因、失败的任务名称。并且对于失败的任务，管理员可以重启这个采集任务。

3. 数据抓取

数据抓取功能是指通过网络爬虫对互联网页面进行抓取，并对具备固定网页格式的页面从中抽取所需要的结构化数据。数据抓取功能支持与数据源网站的比对功能，当数据源网站已发布的数据变更时，它可以对已抓取数据进行更新的功能。

8.5.3　大数据云平台

经各种方式获取的数据，按照数据模型集中存储到大数据云平台上，构建政府和企业的数据

层。大数据云平台是建立在 HDFS、HBase、关系型数据库（如：Oracle）和传统文件系统之上，提供高可靠性、高性能、行列存储、可伸缩、实时读写的大数据存储和管理平台。一个表可以有上亿行，上万列，并且基于 Hadoop 可提供面向列的存储、权限控制和独立检索。依靠 Hadoop 的横向扩展功能，大数据云平台通过不断增加廉价的商用服务器来增加计算和存储能力。

大数据云平台支持 PB 级的数据规模，具有以下特点：

- 高扩展性：可以方便快速地扩展到数以千计的节点（服务器）中。
- 高效性：能够在节点之间动态地移动数据，并保证各个节点的动态平衡，因此处理速度非常快。
- 高容错性：能够自动保存数据的多个副本。
- 低成本：与基于 IBM、HP 的大数据系统相比，基于 EasyDoop 的实施成本大大降低。

8.5.4 大数据服务平台

大数据服务平台提取平台中的结构化和非结构化数据，基于数据模型封装数据服务，发布数据服务接口（API），以统一接口服务方式实现应用系统对大数据平台的数据访问和处理。规范的 API 接口实现了应用层与数据层的松耦合，确保当底层数据库或数据源类型发生变化时，应用层代码无须进行改动。

如图 8-3 所示，除了 API，大数据服务平台还提供了通用型数据服务系统、数据目录管理与服务系统、数据分析系统。

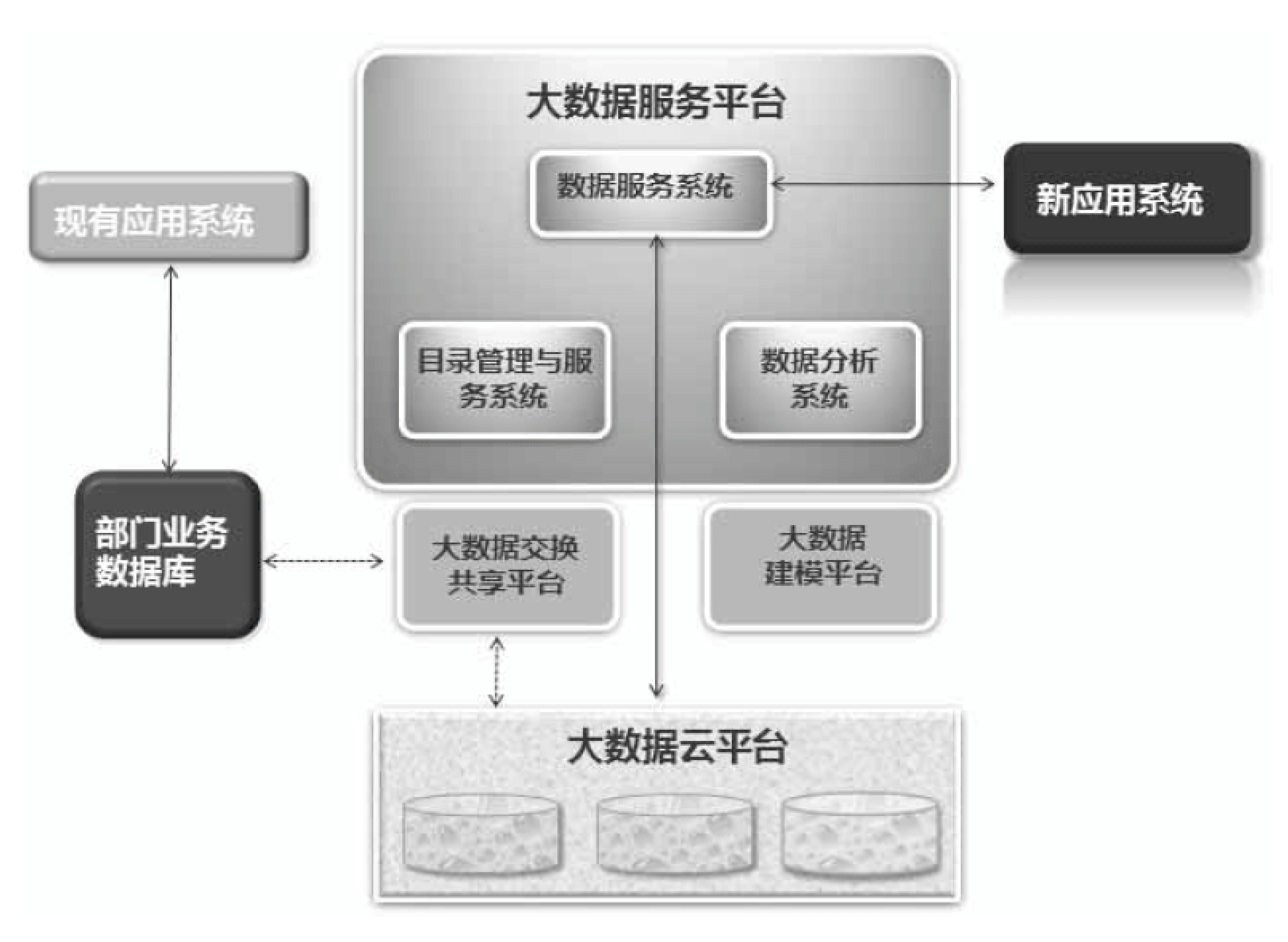


图 8-3 大数据服务平台架构

1. 数据服务系统

数据服务系统提供网页版通用数据管理界面，集中管理平台上的所有结构化和非结构化数据。

- 实现所有数据的聚合展示和查询，包括非结构化数据的统一展示。
- 提供数据各个版本的查询功能。
- 实现数据的增删改功能。
- 按照业务处理流程实现数据的统一流转和处理，业务状态查询。
- 展示数据的生命周期。
- 实现数据的多维归类。
- 实现数据的只读、锁定等设置。

2. 数据目录管理与服务系统

数据目录服务系统采用一种非落地的信息共享模式，是对数据交换共享模式的补充。在目录式共享中，各应用部门（单位）对各自共享的资源有完整的控制权，可有效地解决交换模式中各应用部门（单位）不愿意批量提供数据的问题。

数据目录管理系统是以目录方式实现数据共享，是政府实现信息资源共享的有效手段，使用目录体系可以以更加灵活的方式实现更多部门、更多应用、更多资源的接入与共享。数据目录管理系统的架构如图 8-4 所示。

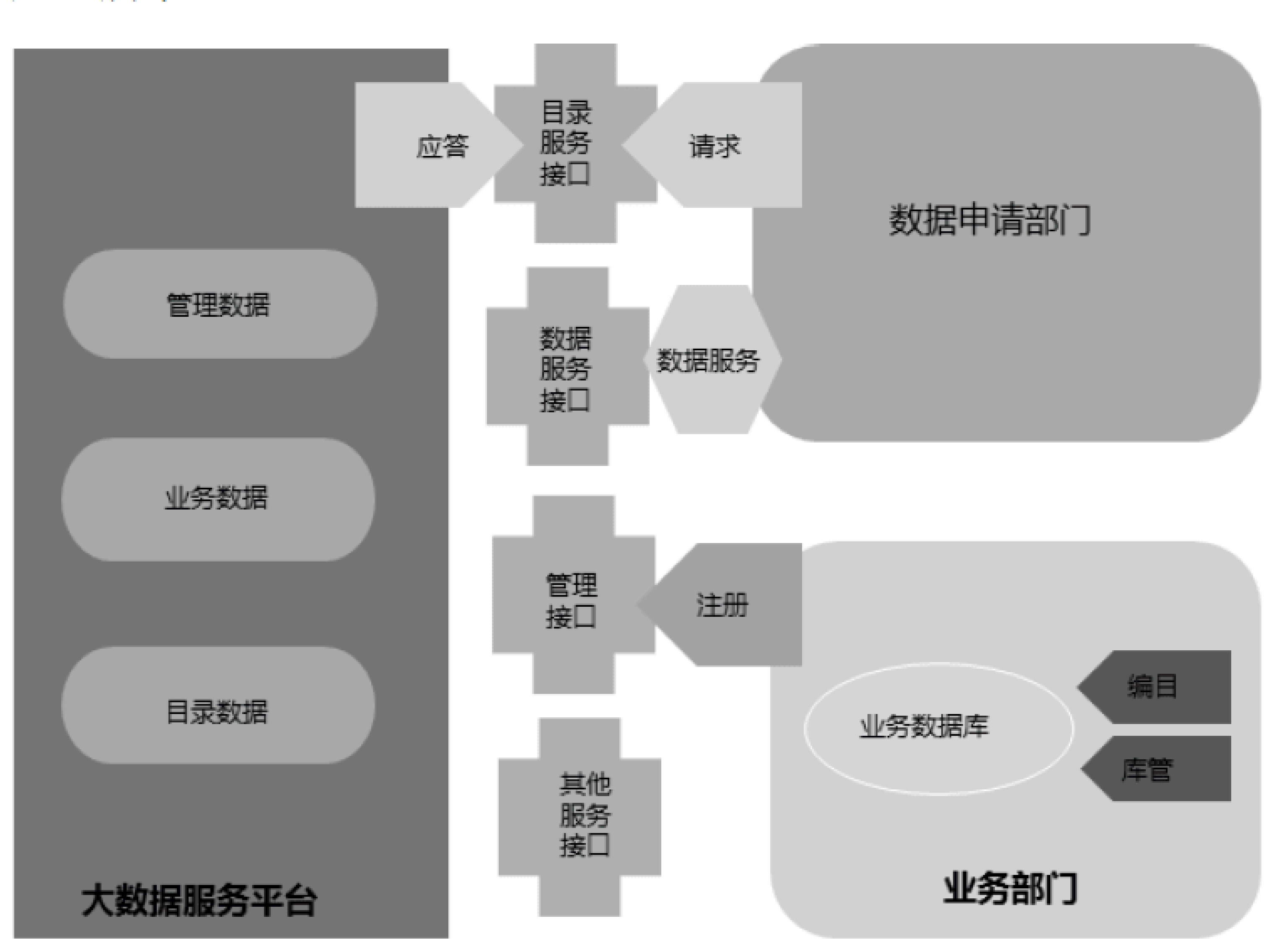


图 8-4　数据目录管理系统

数据目录管理系统的特征是：

- 严格的资源权限管理，不同授权等级的人员看到不同的资源。
- 提供数据目录展示门户系统，可按照应用部门和主题查询。

- 支持多级中心目录管理与访问，支持集中式存储和分布式存储两种方式，可实现横向管理与纵向管理。
- 根据目录内容提供标准接口，提供资源的二次开发支持，为其他应用提供接口查询目录对应资源内容，供给其他应用使用。

3. 数据分析系统

数据分析系统是以案例（Case）为中心展开的。案例就是相关数据集合的对象，是某类待分析目标在数据层面上的多维度体现。数据分析系统提供了管理界面，用于创建一个或多个案例（类似一个文件夹），指定案例下所关联的不同的查询条件（每个查询条件有一个名字，可以认为是一个维度，功能上类似一个子文件夹）。可视化界面可以呈现一个案例下不同维度的数据数量的变迁，并以案例为基本对象进入业务处理流程，这个处理流程可以包含特定行业相关的数据分析。

8.5.5 EasyDoop 平台技术原理分析

图 8-5 显示了 EasyDoop 数据流的过程。EasyDoop 在 Hadoop 集群上存储结构化数据和非结构化数据，异构数据源通过交换和共享平台把数据汇集到 Hadoop 集群上。EasyDoop 提供了数据建模工具，通过 EasyDoop API 按照数据模型将结构化和非结构化数据整合在一个对象中，由行业大数据应用系统进行处理和分析。EasyDoop 还提供了数据安全性机制。

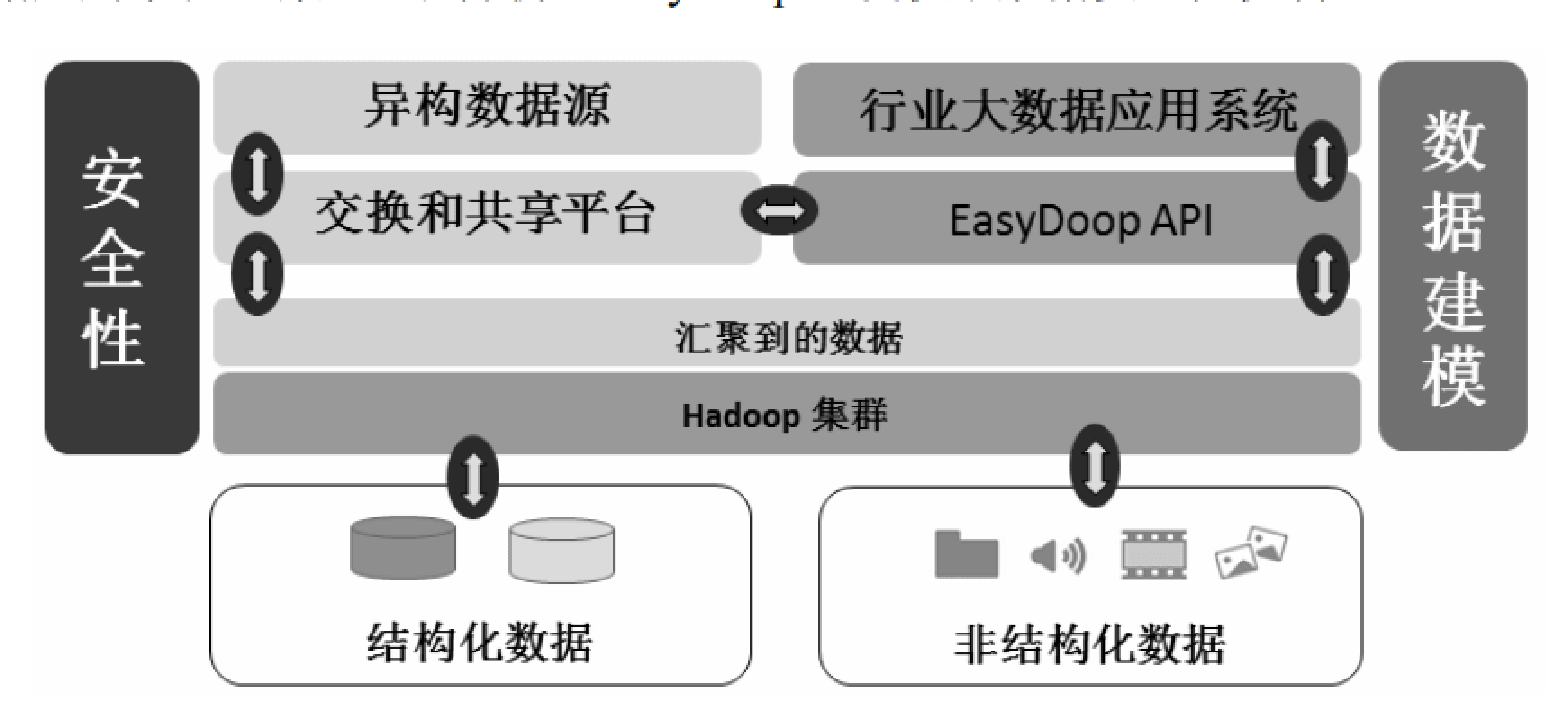


图 8-5　EasyDoop 数据流

1. 数据模型

对于企业或政府部门而言，现有的应用系统就像一个个互不交叉的烟囱。但是，每一个系统产生的数据不应该是割裂的。在大数据背景下我们不能产生更多的烟囱出来，这就需要一个逻辑数据结构把不同的数据联系起来。另外，我们原来的数据更多的是侧重在定量的数据。现在同一份数据，可能既有定量又有定性，所以我们通过一个逻辑数据结构添加新的属性。还有一点，数据是有版本化的，有生命周期和有效期，有些数据 3 年有效，有些 7 年有效，这些可通过数据归档自动完成。定义逻辑结构就是一个数据建模的过程。只有建模之后，才能从不同的烟囱上采集

不同数据到统一的大数据平台上，才能设置数据的归档设置等属性。

EasyDoop 将这些规模日益庞大的源数据汇聚至 Hadoop 时，使用了全平台统一的逻辑数据结构（EasyDoop 数据模型），基于统一的数据模型来规划不同类型数据的访问控制列表、自动业务处理流程、是否在平台上加密等一系列规则。如图 8-6 所示，EasyDoop 数据查询和处理 API 基于统一数据结构来操纵 Hadoop 上的数据。

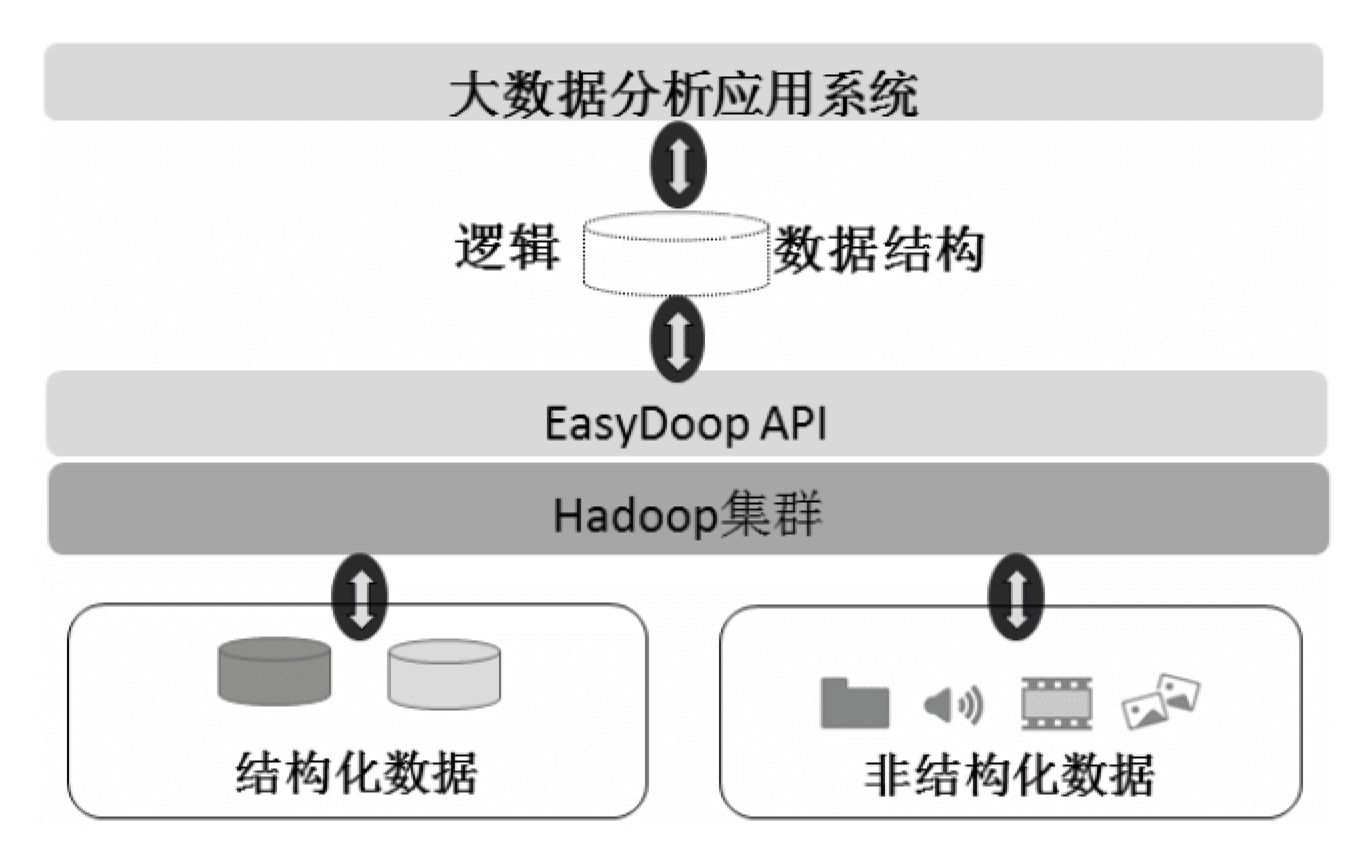


图 8-6 逻辑数据结构

EasyDoop 的建模平台具有以下特征：

- 建模平台支持混合环境，能够同时支持 Hadoop（HDFS 和 HBase）、关系型数据库（IBM DB2、Orale 和 SQL Server）和传统文件系统来存储和读取数据；比如：用户可选择将结构化数据存放在 Oracle 上，同时将非结构化数据存放在 HDFS 上。这为用户在物理数据的底层结构选择上提供了最强的灵活性。
- 提供统一的数据模型，将非结构化数据（文本、语音、流数据等）与结构化数据（元数据）汇聚成统一的信息逻辑对象、从而方便对结构化和非结构化数据加以一致性处理，并统一管理元数据，提供统一标准化字段。
- 提供统一的业务流程模型。将数据分析与业务流程加以绑定，从而允许二者以无须人为干预的自动化方式实现大数据处理。
- 建模平台支持对不同数据模型设置不同的安全访问列表、加密方式、自动业务处理流程、归档时间和归档操作、版本化处理等。
- 建模平台支持自动链接设置，从而为一个数据自动设置多维参数。
- 建模平台的行业数据模型符合国家标准，支持核心元数据的动态扩展。
- 数据模型支持 Spark DataFrame。

2. 大数据交换和共享

除了政府和企业自身的众多 IT 系统上的数据需要采集之外，移动设备以及未来更多的智能穿戴设备上的数据也需要采集。物联网设备，如环保的监测设备，也都是数据的来源。这使得信息的来源变得无时无地不在发生的，并且它是跨平台的，涉及多种设备和多种系统。这种数据源

的多样性，增加了数据采集的难度。

Gartner 认为，透过整合并综合分析各式各样的数据，企业能够取得最独特的商业洞见，并达成流程与决策的极佳化。尽管大多数大数据的运用聚焦于资料取得的多寡和速度，Gartner 调查显示，最终的决胜点取决于能否扩大数据源。

如图 8-7 所示，EasyDoop 的大数据交换和共享平台就是支持爆发式增长的可用数据源，其中包括对社交媒体数据的采集能力。EasyDoop 把数据采集抽象为如下的步骤，并在数据交换和共享平台上实现：

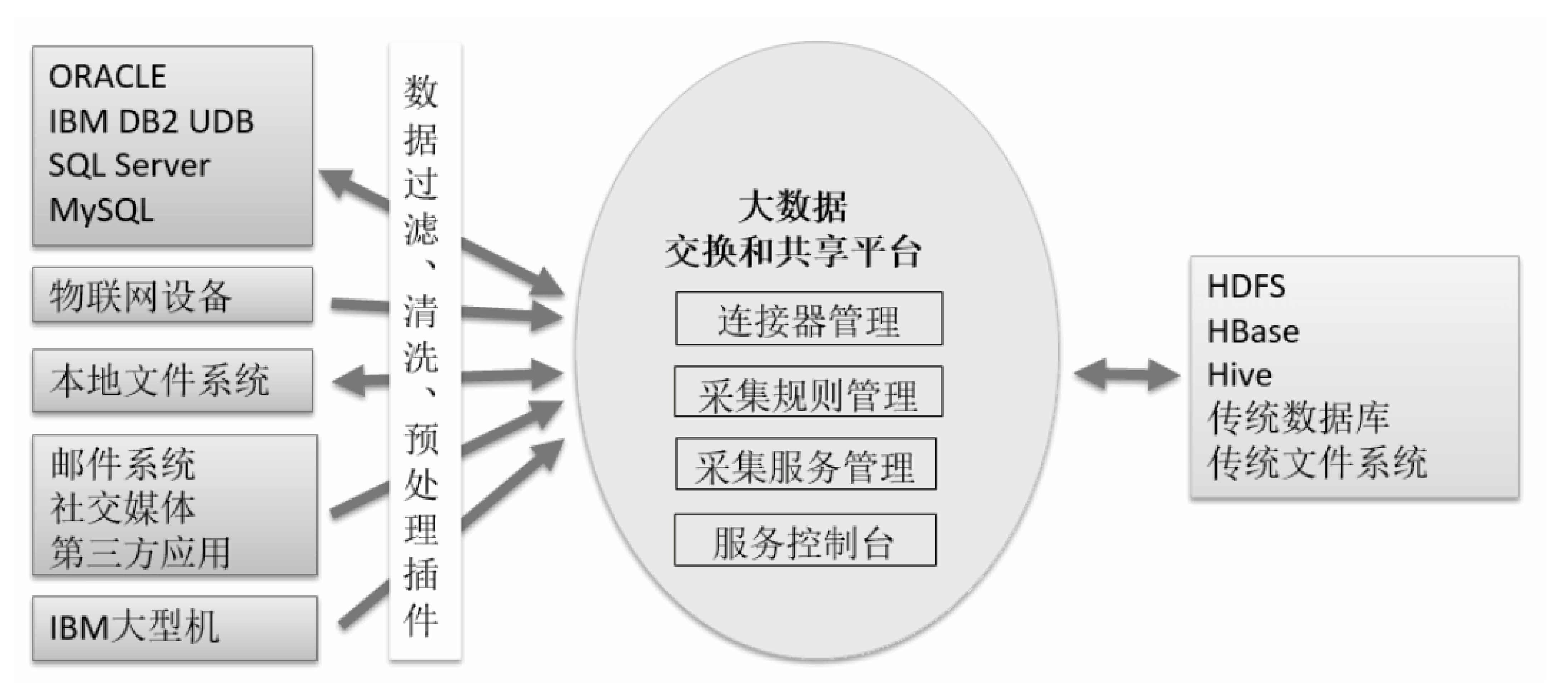


图 8-7　大数据交换和共享

（1）定义采集规则，包括数据源和平台之间的数据映射，数据源的位置信息等。

（2）定义数据源连接器，这包括常见的文件系统连接器、数据库连接器、邮件系统连接器、设备监听器、Web 服务采集器。还包括了只能经由第三方系统（私有系统）API 加以访问的连接器。

（3）定义采集服务。采集服务是一个采集规则、源数据连接器和采集相关配置信息的集合体。与业务流程相结合，采集服务还可以包括保证数据质量的数据比对、清洗、转换、整合、版本化和异常处理等相关操作。

（4）采集管理控制台就是启动、停止和监控所有采集服务的仪表盘。

数据交换和共享平台是一个分布式、高可靠性、高可用性的数据采集平台，是大数据的基础支撑平台，在 Hadoop 层面集成了如下组件：

- 基于 Flume 实现了文件系统和端口等的数据采集。
- 基于 Sqoop 实现了数据库数据采集。
- 基于 Kafka 提供了一个高吞吐和支持缓存的消息队列。
- 基于 Storm 实现了数据流的处理（偏重不同的业务处理需求）。
- 基于 Spark 实现了流式数据的采集。

3. 大数据服务和分析

大数据服务平台提供统一的接口和逻辑数据结构来访问和操控平台上所有数据，为数据的离线/在线分析和挖掘提供基于 Hive 和 Spark 的服务接口。大数据服务平台提供了所有数据的编目列表和统一的数据操纵界面，并提供对数据的申请、审批和接口开放等数据服务功能。大数据服务平台包含了目录管理与服务系统、数据服务系统和数据分析系统。

大数据服务平台集成了 Hadoop 的如下组件：

- 基于 Phoenix 提供了针对 HBase 的 SQL 操作。
- 基于 HDFS API 提供了针对 HDFS 的操纵。
- 基于 Hive 和 Spark 提供了数据统计分析的 SQL 接口。

第 9 章

◀ Spark技术 ▶

Apache Spark 是一个新兴的大数据处理通用引擎，提供了分布式的内存抽象。Spark 最大的特点就是快（Lightning-Fast），可比 Hadoop MapReduce 的处理速度快 100 倍。此外，Spark 提供了简单易用的 API，几行代码就能实现 WordCount。本章介绍 Spark 的框架，Spark Shell 、RDD、Spark SQL、Spark Streaming 等的基本使用。

9.1 Spark 框架

Spark 作为新一代大数据快速处理平台，集成了大数据相关的各种能力。Hadoop 的中间数据需要存储在硬盘上，这产生了较高的延迟。而 Spark 基于内存计算，解决了这个延迟的速度问题。Spark 本身可以直接读写 Hadoop 上任何格式数据，这使得批处理更加快速。

图 9-1 是以 Spark 为核心的大数据处理框架。最底层为大数据存储系统，如：HDFS、HBase 等。在存储系统上面是 Spark 集群模式（也可以认为是资源管理层），这包括 Spark 自带的独立部署模式、YARN 和 Mesos 集群资源管理模式，也可以是 Amazon EC2。Spark 内核之上是为应用提供各类服务的组件。Spark 内核 API 支持 Java、Python、Scala 等编程语言。Spark Streaming 提供高可靠性、高吞吐量的实时流式处理服务，能够满足实时系统要求；MLib 提供机器学习服务，Spark SQL 提供了性能比 Hive 快了很多倍的 SQL 查询服务，GraphX 提供图计算服务。

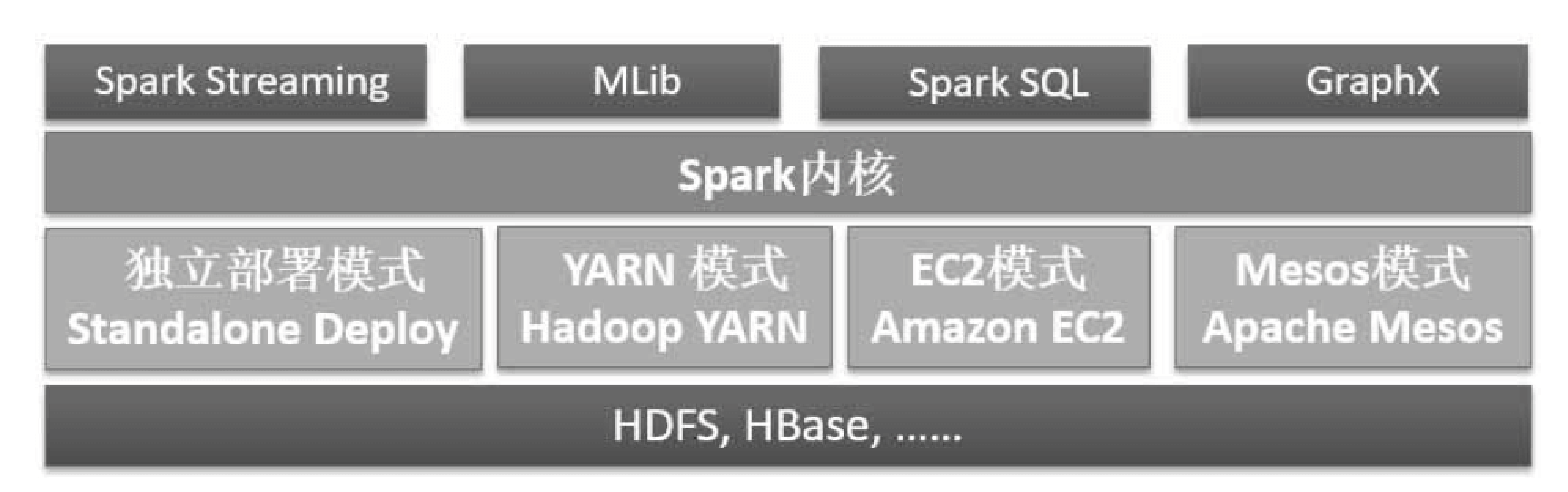


图 9-1　Spark 框架

从上图看出，Spark 有效集成了 Hadoop 组件，可以基于 Hadoop YARN 作为资源管理框架，并从 HDFS 和 HBase 数据源上读取数据。YARN 是 Spark 目前主要使用的资源管理器。Hadoop

能做的，Spark 基本都能做，而且做的比 Hadoop 好。Spark 依然是 Hadoop 生态圈的一员，它替换的主要是 MR 的计算模型而已。资源调度依赖于 YARN，存储则依赖于 HDFS。

Spark 的大数据处理平台是建立在统一抽象的 RDD 之上。RDD 是弹性分布式数据集（Resilient Distributed Dataset）的英文简称，它是一种特殊数据集合，支持多种来源，有容错机制，可以被缓存，支持并行操作。Spark 的一切都是基于 RDD 的。RDD 就是 Spark 输入的数据。

Spark 应用程序在集群上以独立进程集合的形式运行。如图 9-2 所示，主程序（叫做 Driver 程序）中的 SparkContext 对象协调 Spark 应用程序。SparkContext 对象首先连接到多种集群管理器（如：YARN），然后在集群节点上获得 Executor。SparkContext 把应用代码发给 Executor，Executor 负责应用程序的计算和数据存储。

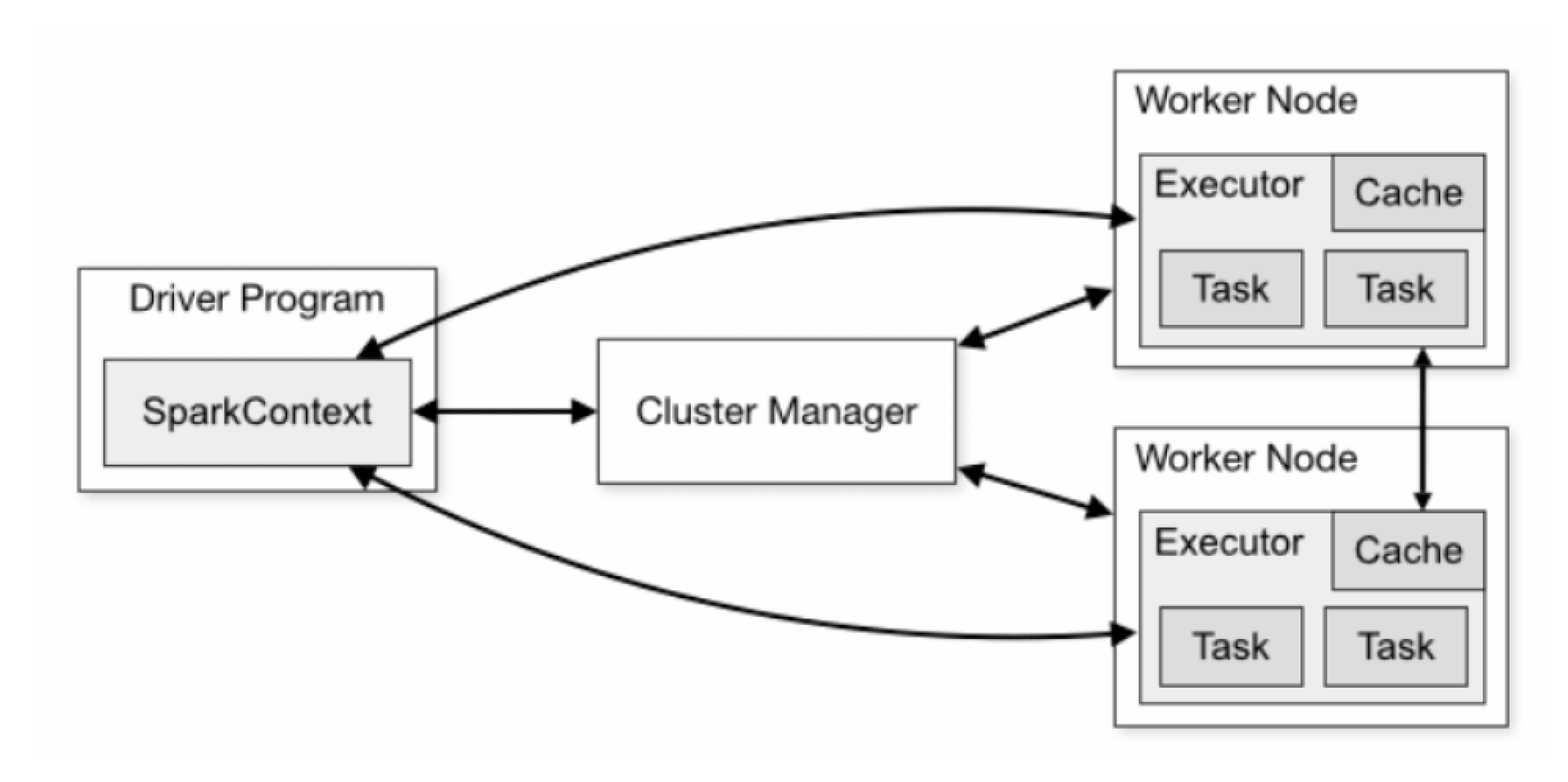


图 9-2　集群模式

每个应用程序都拥有自己的 Executor。Executor 为应用程序提供了一个隔离的运行环境，以 Task 的形式执行作业。对于 Spark Shell 来说，这个 Driver 就是与用户交互的进程。

9.1.1　安装 Spark

最新的 Spark 版本是 1.6.1。它可以运行在 Windows 或 Linux 机器上。运行 Spark 需要 Java JDK 1.7，CentOS 6.x 系统默认只安装了 Java JRE，还需要安装 Java JDK，并确保配置好 JAVA_HOME、PATH 和 CLASSPATH 变量。此外，Spark 会用到 HDFS 与 YARN，因此读者要先安装好 Hadoop。我们可以从 Spark 官方网站 http://spark.apache.org/downloads.html 上下载 Spark，如图 9-3 所示。

Download Apache Spark™

Our latest version is Spark 1.6.1, released on March 9, 2016 (release notes) (git tag)

1. Choose a Spark release: 1.6.1 (Mar 09 2016)
2. Choose a package type: Pre-build with user-provided Hadoop [can use with most Hadoop distributions]
3. Choose a download type: Direct Download
4. Download Spark: spark-1.6.1-bin-without-hadoop.tgz
5. Verify this release using the 1.6.1 signatures and checksums.

Note: Scala 2.11 users should download the Spark source package and build with Scala 2.11 support.

图 9-3　下载安装包

有几种 Package type，分别为：

- Source code：Spark 源码，需要编译才能使用。
- Pre-build with user-provided Hadoop：“Hadoop free”版，可应用到任意 Hadoop 版本。
- Pre-build for Hadoop 2.6 and later：基于 Hadoop 2.6 的预编译版，需要与本机安装的 Hadoop 版本对应。可选的还有 Hadoop 2.4 and later、Hadoop 2.3、Hadoop 1.x，以及 CDH 4。

本书选择的是 Pre-build with user-provided Hadoop，简单配置后可应用到任意 Hadoop 版本。下载后，执行如下命令进行安装：

```
sudo tar -zxf spark-1.6.1-bin-without-hadoop.tgz -C /usr/local/
cd /usr/local
sudo mv ./spark-1.6.1-bin-without-hadoop/ ./spark
sudo chown -R hadoop:hadoop ./spark
```

9.1.2　配置 Spark

安装后，进入 conf 目录，以 spark-env.sh.template 文件为模块创建 spark-env.sh 文件，然后修改其配置信息，命令如下：

```
cd /usr/local/spark
cp ./conf/spark-env.sh.template ./conf/spark-env.sh
```

编辑 ./conf/spark-env.sh（vim ./conf/spark-env.sh），在文件的最后加上如下一行：

```
export SPARK_DIST_CLASSPATH=$(/usr/local/hadoop/bin/hadoop classpath)
```

保存后，Spark 就可以启动和运行了。在 ./examples/src/main 目录下有一些 Spark 的示例程序，有 Scala、Java、Python、R 等语言的版本。我们可以先运行一个示例程序 SparkPi（即

计算 π 的近似值），执行如下命令：

```
cd /usr/local/spark
./bin/run-example SparkPi
```

执行时会输出非常多的运行信息，输出结果不容易找到，可以通过 grep 命令进行过滤（命令中的 2>&1 可以将所有的信息都输出到 stdout 中）：

```
./bin/run-example SparkPi 2>&1 | grep "Pi is roughly"
```

过滤后的运行结果为 π 的 5 位小数近似值 。

9.2 Spark Shell

以前的统计和机器学习依赖于数据抽样。从统计的角度来看，抽样如果足够随机，其实可以很精准地反应全集的结果，但事实上往往很难做到随机，所以通常做出来也会不准。现在大数据解决了这个问题，它不是通过优化抽样的随机来解决，而是通过全量数据来解决。要解决全量的数据就需要有强大的处理能力，Spark 首先具备强大的处理能力，其次 Spark Shell 带来了即席查询。做算法的工程师，以前经常是在小数据集上跑个单机，然后看效果不错，一到全量上，就可能和单机效果很不一样。有了 Spark 后就不一样了，尤其是有了 Spark Shell。可以边写代码，边运行，边看结果。Spark 提供了很多的算法，最常用的是贝叶斯、word2vec、线性回归等。作为算法工程师，或者大数据分析师，一定要学会用 Spark Shell。

Spark Shell 提供了简单的方式来学习 Spark API，也提供了交互的方式来分析数据。Spark Shell 支持 Scala 和 Python，本书选择使用 Scala 来进行介绍。Scala 集成了面向对象和函数语言的特性，并运行于 Java 虚拟机之上，兼容现有的 Java 程序。Scala 是 Spark 的主要编程语言，如果仅仅是写 Spark 应用，并非一定要用 Scala，用 Java 和 Python 都是可以的。使用 Scala 的优势是开发效率更高，代码更精简，并且可以通过 Spark Shell 进行交互式实时查询，方便排查问题。执行如下命令启动 Spark Shell：

```
./bin/spark-shell
```

启动成功后会有“scala >”的命令提示符。这表明已经成功启动了 Spark Shell。在 Spark Shell 启动时，输出日志的最后有这么几条信息：

```
16/04/16 17:25:47 INFO repl.SparkILoop: Created spark context..
Spark context available as sc.
```

这些信息表明 SparkContext 已经初始化好了，可通过对应的 sc 变量直接进行访问。Spark 的主要抽象是分布式的数据集合 RDD，它可被分发到集群各个节点上，进行并行操作。一个

RDD 可以通过 Hadoop InputFormats 创建（如 HDFS），或者从其他 RDDs 转化而来。下面我们从 ./README 文件新建一个 RDD，代码如下：

```
scala>val textFile = sc.textFile("file:///usr/local/spark/README.md")
```

上述的 sc 是 Spark 创建的 SparkContext，我们使用 SparkContext 对象加载本地文件 README.md 来创建 RDD。输出结果如下：

```
textFile: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[1] at
textFile at <console>:27
```

上述返回结果为一个 MapPartitionsRDD 文件。需要说明的是，加载 HDFS 文件和本地文件都是使用 textFile ，区别在于前缀“hdfs://”为 HDFS 文件，而“file:// ”为本地文件。上述代码中通过“file://”前缀指定读取本地文件，直接返回 MapPartitionsRDD。Spark Shell 默认方式是读取 HDFS 中的文件。从 HDFS 读取的文件先转换为 HadoopRDD，然后隐式转换成 MapPartitionsRDD。

上面的例子使用 Spark 中的文本文件 README.md 创建一个 RDD textFile，文件中包含了若干文本行。将该文本文件读入 RDD textFile 时，其中的文本行将被分区，以便能够分发到集群中并行化操作。我们可以想象，RDD 有多个分区，每个分区上有多行的文本内容。RDDs 支持两种类型的操作：

- actions：在数据集上运行计算后返回结果值。
- transformations：转换。从现有 RDD 创建一个新的 RDD。

下面我们演示 count()和 first()操作：

```
scala>textFile.count()      // RDD 中的 item 数量，对于文本文件，就是总行数
```

输出结果为：

```
res0: Long = 95

scala>textFile.first()    // RDD 中的第一个 item，对于文本文件，就是第一行内
容
```

输出结果为：

```
res1: String = # Apache Spark
```

上面这两个例子都是 action 的例子。接着演示 transformation，通过 filter transformation 来筛选出包含 Spark 的行，返回一个新的 RDD，代码如下：

```
scala>val linesWithSpark = textFile.filter(line =>
line.contains("Spark"))
```

```
scala>linesWithSpark.count()        // 统计行数
```

上面的 linesWithSpark RDD 有多个分区，每个分区上只有包含了 Spark 的若干文本行。输出结果为：

```
res4: Long = 17
```

上述结果表明一共有 17 行内容包含“Spark”，这与通过 Linux 命令 cat ./README.md | grep "Spark" -c 得到的结果一致，说明是正确的。action 和 transformation 可以用链式操作的方式结合使用，使代码更为简洁：

```
scala>textFile.filter(line => line.contains("Spark")).count()     // 统计包含 Spark 的行数
```

RDD 的 actions 和 transformations 可用在更复杂的计算中。例如，通过如下代码可以找到包含单词最多的那一行内容共有几个单词：

```
scala>textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
```

输出结果为：

```
res5: Int = 14
```

上述代码将每一行文本内容使用 split 进行分词，并统计分词后的单词数。将每一行内容 map 为一个整数，这将创建一个新的 RDD，并在这个 RDD 中执行 reduce 操作，找到最大的数。map()、reduce()中的参数是 Scala 的函数字面量（function literals），并且可以使用 Scala/Java 的库。例如，通过使用 Math.max() 函数（需要导入 Java 的 Math 库），可以使上述代码更容易理解：

```
scala>import java.lang.Math
scala>textFile.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b))
```

词频统计（WordCount）是 Hadoop MapReduce 的入门程序，Spark 可以更容易地实现。首先结合 flatMap、map 和 reduceKey 来计算文件中每个单词的词频：

```
scala>val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)
```

输出结果为(string，int)类型的键值对 ShuffledRDD。这是因为 reduceByKey 操作需要进行 Shuffle 操作，返回的是一个 Shuffle 形式的 ShuffleRDD：

```
wordCounts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:29
```

然后使用 collect 聚合单词计算结果：

```
scala>wordCounts.collect()
```

输出结果为：

```
res7: Array[(String, Int)] = Array((package,1), (For,2), (Programs,1),
(processing,1), (Because,1), (The,1)...)
```

Spark 支持将数据缓存在集群的内存缓存中，当数据需要反复访问时这个特征非常有用。调用 cache()，就可以将数据集进行缓存：

```
scala>textFilter.cache()
```

9.3 Spark 编程

无论 Windows 或 Linux 操作系统，都是基于 Eclipse 或 Idea 构建开发环境，通过 Java、Scala 或 Python 语言进行开发。根据开发语言的不同，我们需要预先准备好 JDK、Scala 或 Python 环境，然后在 Eclipse 中下载安装 Scala 或 Python 插件。

下面我们通过一个简单的应用程序 SimpleApp 来演示如何通过 Spark API 编写一个独立应用程序。不同于使用 Spark Shell 自动初始化的 SparkContext，独立应用程序需要自己初始化一个 SparkContext，将一个包含应用程序信息的 SparkConf 对象传递给 SparkContext 构造函数。对于独立应用程序，使用 Scala 编写的程序需要使用 sbt 进行编译打包，相应地，Java 程序使用 Maven 编译打包，而 Python 程序通过 spark-submit 直接提交。

在终端中执行如下命令，创建一个文件夹 sparkapp 作为应用程序根目录：

```
cd ~                    # 进入用户主文件夹
mkdir ./sparkapp          # 创建应用程序根目录
mkdir -p ./sparkapp/src/main/scala      # 创建所需的文件夹结构
```

9.3.1 编写 Spark API 程序

在 ./sparkapp/src/main/scala 下建立一个名为 SimpleApp.scala 的文件（vim ./sparkapp/src/main/scala/SimpleApp.scala），添加代码如下：

```
/* SimpleApp.scala */
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf
```

```
object SimpleApp {
  //使用关键字 def 声明函数，必须为函数指定参数类型
  def main(args: Array[String]) {
    val logFile = "file:///usr/local/spark/README.md" // 一个本地文件
    //创建 SparkConf 对象，该对象包含应用程序的信息
    val conf = new SparkConf().setAppName("Simple Application")
    //创建 SparkContext 对象，该对象可以访问 Spark 集群
    val sc = new SparkContext(conf)
    val logData = sc.textFile(logFile, 2).cache()
     //line=>line.contains(..)是匿名函数的定义，line 是参数
    val numAs = logData.filter(line => line.contains("a")).count()
    val numBs = logData.filter(line => line.contains("b")).count()
    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))
  }
}
```

上述程序计算 /usr/local/spark/README 文件中包含 “a” 的行数和包含 “b” 的行数。不同于 Spark Shell，独立应用程序需要通过“val sc = new SparkContext(conf)”初始化 SparkContext，SparkContext 的参数 SparkConf 包含了应用程序的信息。

9.3.2　使用 sbt 编译并打成 jar 包

该程序依赖 Spark API，因此我们需要通过 sbt（或 mvn）进行编译打包。我们以 sbt 为例，创建一个包含应用程序代码的 jar 包。在 ./sparkapp 中新建文件 simple.sbt（vim ./sparkapp/simple.sbt），添加如下内容，声明该独立应用程序的信息以及与 Spark 的依赖关系：

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.10.5"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.6.1"
```

文件 simple.sbt 需要指明 Spark 和 Scala 的版本。上述版本信息可以从 Spark Shell 获得。我们启动 Spark Shell 的过程中，当输出到 Spark 的符号图形时，可以看到相关的版本信息。

Spark 中没有自带 sbt，需要手动安装 sbt，我们选择安装在/usr/local/sbt 中：

```
sudo mkdir /usr/local/sbt
sudo chown -R hadoop /usr/local/sbt      # 此处的 hadoop 为你的用户名
cd /usr/local/sbt
```

下载 sbt 后，拷贝至 /usr/local/sbt 中。接着在 /usr/local/sbt 中创建 sbt 脚本（vim ./sbt），添加如下内容：

```
#!/bin/bash
SBT_OPTS="-Xms512M -Xmx1536M -Xss1M -XX:+CMSClassUnloadingEnabled -
XX:MaxPermSize=256M"
java $SBT_OPTS -jar `dirname $0`/sbt-launch.jar "$@"
```

保存后，为 ./sbt 脚本增加可执行权限：

```
chmod u+x ./sbt
```

最后检验 sbt 是否可用：

```
./sbt sbt-version
```

只要能得到版本信息就说明 sbt 安装没问题了。接着，我们可以通过如下代码将整个应用程序打包成 JAR 文件：

```
/usr/local/sbt/sbt package
```

打包成功的话，会输出“Done Packaging”信息。生成的 jar 包的位置为 ~/sparkapp/target/scala-2.10/simple-project_2.10-1.0.jar。

9.3.3 运行程序

一旦应用程序被打包成 jar 文件，就可以通过 /bin/spark-submit 脚本启动应用程序。将生成的 jar 包通过 spark-submit 提交到 Spark 中运行了，命令如下：

```
/usr/local/spark/bin/spark-submit --class "SimpleApp" \
~/sparkapp/target/scala-2.10/simple-project_2.10-1.0.jar
```

如果你觉得输出信息太多，可以通过如下命令过滤结果信息：

```
/usr/local/spark/bin/spark-submit --class "SimpleApp" \
~/sparkapp/target/scala-2.10/simple-project_2.10-1.0.jar 2>&1 | grep
"Lines with a:"
```

最终得到的结果如下：

```
Lines with a: 58, Lines with b: 26
```

9.4 RDD

上面几节描述了 Spark 程序的基本步骤，让读者有了一个真实的体验。由于 Spark 一切都是基于 RDD 的，RDD（Resilient Distributed Datasets，弹性分布式数据集）就是 Spark 输入的数据。本节我们首先阐述 RDD 的由来，然后详细阐述 RDD 的创建和操作。

我经常在想，Spark 当初为何提出 RDD 概念，相对于 Hadoop，RDD 给 Spark 带来何等优势？我们知道，对于 Hadoop 中一个独立的计算，例如在一个迭代过程中，除文件系统

（HDFS）外没有提供其他存储的概念。两个 MapReduce 作业之间数据共享只有一个办法，就是将其写到一个外部存储系统，如分布式文件系统。这会引起大量的开销，拉长应用的执行时间。所以，如果在计算过程中能共享数据，那将会降低集群开销，同时还能减少任务执行时间。而 Spark 中的 RDD 就是让用户可以直接控制数据的共享。RDD 具有可容错和并行数据结构特征，可以指定数据存储到硬盘还是内存，控制数据的分区方法，并在数据集上进行丰富的操作。当一个 RDD 的某个分区丢失的时候，RDD 有足够的信息记录其如何通过其他的 RDD 进行计算，且只需重新计算该分区。因此，丢失的数据可以很快恢复，而不需要昂贵的复制代价。

Spark 和很多其他分布式计算系统的思想是：把一个超大的数据集，切分成 N 个小堆，找 M 个执行器（M < N），各自拿一块或多块数据操作，操作结果再收集在一起。这个拥有多个分块的数据集就叫 RDD。RDD 是一个分区的只读记录的集合。Spark 在集群中并行地执行任务，并行度由 Spark 中的 RDD 决定。RDD 中的数据被分区存储在集群中（碎片化的数据存储方式），正是由于数据的分区存储使得任务可以并行执行。分区数量越多，并行越高。RDD 具有几个特征：

- 分区（partition）：一个 RDD 有多个分区组成，等于将 RDD 数据切分。分区数据是数据集的原子组成部分，能够进行并行计算。如图 9-4 所示，一个 RDD 有 3 个分区。
- 算子（compute）：用于说明在 RDD 上执行何种计算，图 9-4 描述了 RDD 的多种算子。我们可以简单地把算子等同于函数。

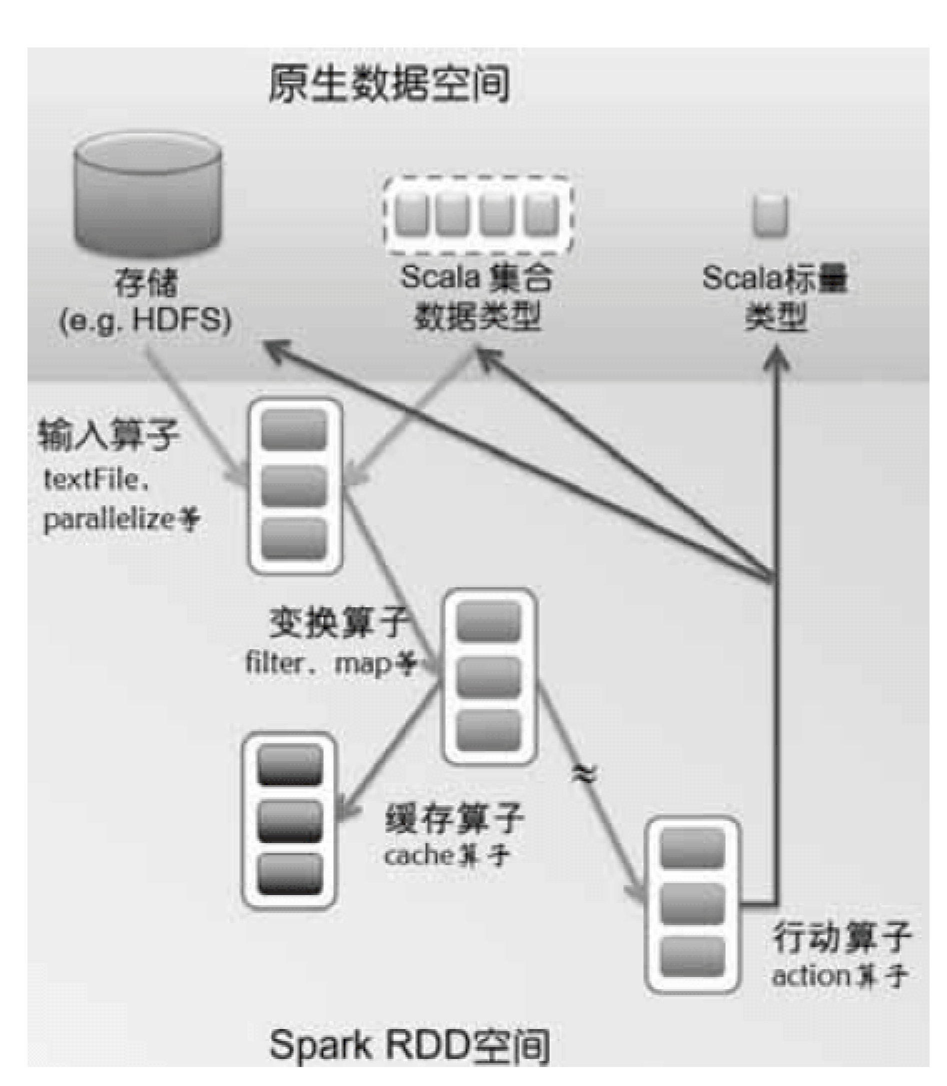


图 9-4　RDD 操作

- 依赖（dependency）：计算每个 RDD 对父 RDD 的依赖列表。下面分小节描述两种依赖关系。

9.4.1　RDD 算子和 RDD 依赖关系

算子是 RDD 中定义的函数，图 9-4 描述了 Spark 在运行过程中通过算子对 RDD 进行创建、

转换（Transformation）和行动（Action）。RDD 转换和行动统称为 RDD 操作。我们按照数据的处理步骤描述各个算子：

（1）输入：在 Spark 程序运行中，数据从外部数据空间（如分布式存储：textFile 读取 HDFS 等，parallelize 方法输入 Scala 集合或数据）输入 Spark，创建了 RDD，数据进入了 Spark RDD 空间。

（2）运行：在 Spark 输入数据形成 RDD 后便可以通过变换算子，如 filter 等，对数据进行操作并将 RDD 转化为新的 RDD，通过 Action 算子，触发 Spark 提交作业。如果数据需要复用，可以通过 Cache 算子，将数据缓存到内存。

（3）输出：程序运行结束后，数据会存储到分布式存储中（如 saveAsTextFile 输出到 HDFS），或 Scala 数据或集合中（collect 输出到 Scala 集合，count 返回 Scala int 型数据）。

需要注意的是，创建 RDD 并不会导致集群执行分布式计算。相反，RDD 只是定义了作为计算过程中间步骤的逻辑数据集，只有调用 RDD 上的 action 时，分布式计算才会真正执行。读者可以形象地理解为，除了 action 之外，前面的代码都只是在定义一个数据处理流的各个步骤（创建 RDD，转换 RDD，等等），只有碰到了 action，才提交给集群上执行这个流程。

如图 9-4 所示，大致可以分为三大类算子:

- Value 数据类型的 Transformation 算子，这种转换并不触发提交作业，针对处理的数据项是 Value 型的数据。
- Key-Value 数据类型的 Transfromation 算子，这种转换并不触发提交作业，针对处理的数据项是 Key-Value 型的数据对。
- Action 算子，这类算子会触发 SparkContext 提交 Job 作业。

为了创建 RDD，可以从外部存储中读取数据，例如从 HDFS 或其他 Hadoop 支持的输入数据格式中读取。也可以通过读取文件、数组或 JSON 格式的数据来创建 RDD。对于应用来说，数据是本地化的，此时你仅需要使用 parallelize 方法，便可以将 Spark 的特性作用于相应数据，并通过 Apache Spark 集群对数据进行并行化分析。比如：

```
   val thingsRDD = sc.parallelize(List("spoon", "fork", "plate", "cup",
"bottle"))
```

运行 Spark 时，需要创建 SparkContext。使用 Spark Shell 交互式命令行时，SparkContext 会自动创建（就是上述命令中的 sc）。当调用 SparkContext 对象的 parallelize 方法后，我们会得到一个经过分区的 RDD，这些数据将被分发到集群的各个节点上。

Spark 的核心数据模型是 RDD，但 RDD 是个抽象类，具体由各子类实现，如 MappedRDD、ShuffledRDD 等子类。Spark 将常用的大数据操作都转化成为 RDD 的子类。图 9-5 显示了 RDD 的一个使用案例。通过 TextFile 从 HDFS 上创建 RDD，然后通过 flatmap、map 和 join 转换，最后通过 saveAsQuenceFile 操作存储在 HDFS 和 HBase 上。当用户对一个 RDD 执行 action（如 count 或 save）操作时，会构建一个由若干阶段（stage）组成的一个 DAG（有向无环图）以执行程序。

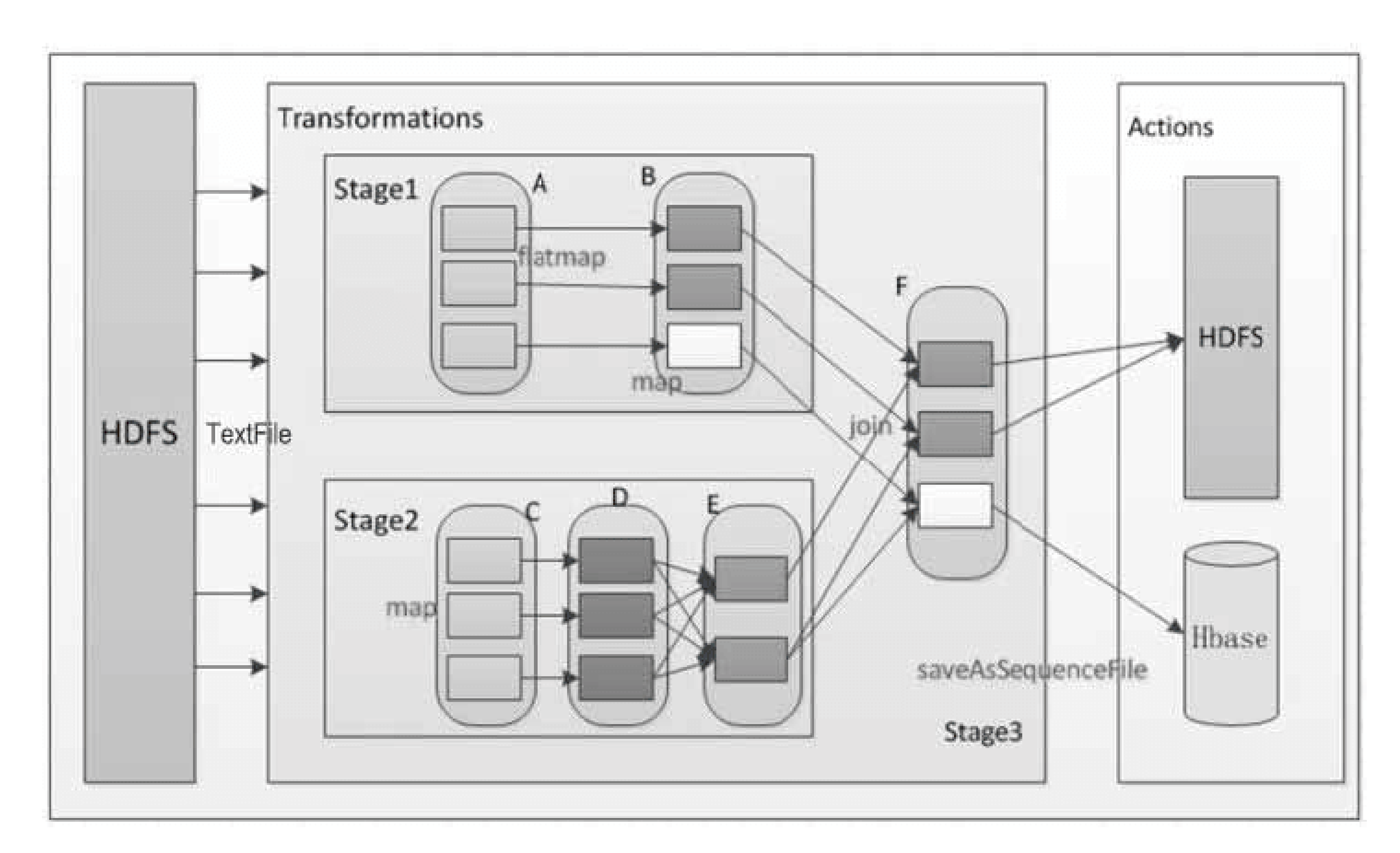


图 9-5 RDD 示例

Spark 中的表示 RDD 之间的依赖关系主要分为两类：

- 窄依赖：父 RDD 的每个分区都至多被一个子 RDD 的分区使用;
- 宽依赖：子 RDD 的每个分区都依赖于所有父 RDD 的所有分区或多个分区，也就是说存在一个父 RDD 的一个分区对应一个子 RDD 的多个分区。

例如，map 操作是一种窄依赖，而 join 操作是一种宽依赖。宽依赖需要所有的父 RDD 数据可用并且数据已经通过类 MapReduce 的操作 shuffle 完成。每个 stage 都包含尽可能多的连续的窄依赖型转换。各个阶段之间的分界则是宽依赖所需的 shuffle 操作。

在创建 RDD 之后，既可以进行数据转换，也可以对其进行 action 操作。这意味着使用 transformation 可以改变数据格式、进行数据查询或数据过滤操作等，使用 action 操作，可以触发数据的改变、抽取数据、收集数据甚至进行计数。

9.4.2 RDD 转换操作

Transformation（转换）是指从已经存在的数据集上创建一个新的数据集，是数据集的逻辑操作，并没有真正计算。我们可以理解转换操作为一种惰性操作，它只是定义了一个新的 RDD，而不是立即计算它。相反，action 操作则是立即计算，并返回结果给程序，或者将结果写入到外部存储中。RDD 转换操作包含了下面两大类转换操作。

1. 基础转换操作

- map(func)：由数据集中的每条元素经过 func 函数转换后组成一个新的分布式数据集;
- filter(func)：过滤作用，选取数据集中让 func 函数返回值为 true 的元素组成一个新的数据集;
- flatMap(func)：类似于 map，但是每一个输入元素，会被映射为 0 到多个输出元素（因此，func 函数的返回值是一个 Seq，而不是单一元素）;
- sample(withReplacement, frac, seed)：根据给定的随机种子 seed，随机抽样出数量为 frac

的数据（可以选择有无替代 replacement）；

- union(otherDataset)：返回一个新的数据集，由原数据集和参数联合而成;
- mapPartitions(func)：类似于 map，但单独运行在 RDD 分区;
- intersection(otherDataset)：返回一个数据集交集元素的新的 RDD;
- distinct([numTasks])：返回一个数据集去重之后的新的数据集。

2. 键-值转换操作

下面的转换操作只能在键-值对形式的 RDD 上执行。最常见的就是 shuffle 操作，通过键进行分组或聚合元素。比如：reduceByKey 操作对文件中每行出现的文字次数进行计算。

- groupByKey([numTasks])：当在一个由键值对（K,V）组成的数据集上调用时，按照 K 进行分组，返回一个（K，Seq[V])对的数据集。注意：默认情况下，输出的并行程度取决于父 RDD 的分区数，但是你可以传入可选的 numTasks 参数来设置不同数目的并行任务。
- reduceByKey(func, [numTasks])：当在一个键值对（K，V)的数据集上调用时，返回一个（K，V）对的数据集，key 相同的值，都被使用指定的 reduce 函数聚合到一起。和 groupbykey 类似，任务的个数是可以通过第二个可选参数来配置的。
- join(otherDataset, [numTasks])：在类型为（K,V)和（K,W)类型的数据集上调用，返回一个（K,(V,W))类型的数据集，每个 key 中的所有元素都在一起的数据集。

9.4.3 RDD 行动（Action）操作

对数据集进行变换操作（如 map 和 filter）而得到一个或多个 RDD，然后调用这些 RDD 的 actions（行动）类的操作。这类操作的目的是返回一个值或是将数据导入到存储系统中。动作类的操作如 count（返回数据集的元素数），collect（返回元素本身的集合）和 save（输出数据集到存储系统）。

虽然 RDD 的转换操作是 RDD 的核心之一，通过转换操作实现不同的 RDD，但是转换操作不会触发 Job 的提交，仅仅是标记对 RDD 的操作，形成 DAG 图，只有遇到一个 Action（行动）操作时才触发 Job 的执行。spark 直到 RDD 第一次调用一个 Action 时才真正计算 RDD。Action 操作包含了下面两类操作。

1. 常用行动操作

- reduce(func)：通过函数 func 聚集数据集中的所有元素。func 函数接受 2 个参数，返回一个值。这个函数必须是关联性的，确保可以被正确地并发执行。
- collect()：在 Driver 的程序中，以数组的形式，返回数据集的所有元素。这通常会在使用 filter 或者其他操作后，返回一个足够小的数据子集再使用。直接将整个 RDD 集 Collect 返回，很可能会让 Driver 程序 OOM。
- count()：返回数据集的元素个数。

- take(n)：返回一个数组，由数据集的前 n 个元素组成。
- first()：返回数据集的第一个元素（类似于 take(1)）。
- foreach(func)：在数据集的每一个元素上，运行函数 func。这通常用于更新一个累加器变量，或者和外部存储系统做交互。

上面的行动操作可以实现大部分 MapReduce 流式计算的任务。

2. 存储行动操作

- saveAsTextFile(path)：将数据集的元素，以 textfile 的形式，保存到本地文件系统、HDFS 或者任何其他 hadoop 支持的文件系统。Spark 将会调用每个元素的 toString 方法，并将它转换为文件中的一行文本。
- saveAsSequenceFile(path)：将数据集的元素，以 sequencefile 的格式，保存到本地系统、HDFS 或者任何其他 hadoop 支持的文件系统的指定的目录下。RDD 的元素必须由 key-value 对组成，并都实现了 Hadoop 的 Writable 接口，或隐式可以转换为 Writable（Spark 包括了基本类型的转换，例如 int、double、string 等等）。

9.4.4　RDD 控制操作

除了 RDD 转换操作和 Action 操作之外，RDD 还有控制操作，比如：缓存操作（cache），释放内存操作（unpersist），checkpoint 操作（直接将 RDD 持久化到磁盘上）。每次进行 action 操作时，例如 count()操作，Spark 将重新启动所有的转换操作，计算将运行到最后一个转换操作，然后 count 操作返回计算结果，这种运行方式速度会较慢。为了解决该问题和提高程序运行速度，可以将 RDD 的数据缓存到内存当中。当你反复运行 action 操作时，能够避免每次计算都从头开始，直接从缓存到内存中的 RDD 得到相应的结果。

如果你想将 RDD 从缓存中清除，可以使用 unpersist()方法。如果不手动删除的话，在内存空间紧张的情况下，Spark 会采用最近最久未使用（least recently used logic，LRU）调度算法删除缓存在内存中最久的 RDD。

9.4.5　RDD 实例

下面我们总结一下 Spark 从开始到结果的运行过程：

步骤 01　创建某种数据类型的 RDD。
步骤 02　对 RDD 中的数据进行转换操作，例如过滤操作。
步骤 03　在需要重用的情况下，对转换后或过滤后的 RDD 进行缓存。
步骤 04　在 RDD 上进行 action 操作，例如提取数据、计数、存储数据到 HDFS 等。

下面给出的是 RDD 的部分转换操作函数：

- filter()
- map()
- sample()
- union()
- groupbykey()
- sortbykey()
- combineByKey()
- subtractByKey()
- mapValues()
- Keys()
- Values()

下面给出的是 RDD 的部分 action 操作：

- collect()
- count()
- first()
- countbykey()
- saveAsTextFile()
- reduce()
- take(n)
- countBykey()
- collectAsMap()
- lookup(key)

下面我们来看几个实例。假定 rdd 为{1,2,3,3}的数据集，rdd1 为{1,2,3}的数据集，rdd2 为{3,4,5}的数据集。表 9-1 所示是一些转换操作的例子。

表 9-1　一些转换操作的例子

函数	例子	结果	说明
map	rdd.map(x=>x+1)	{2,3,4,4}	对 RDD 中每个元素进行加 1 操作
flatMap	rdd.flatMap(x=>x.to(3))	{1,2,3,2,3,3,3}	遍历当前每个元素，然后生成从当前元素到 3 的集合
filter	rdd.filter(x=>x!=1)	{2,3,3}	过滤 RDD 中不等于 1 的元素
distinct	rdd.distinct()	{1,2,3}	对 RDD 元素去重
union	rdd1.union(rdd2)	{1,2,3,3,4,5}	返回两个 RDD 的并集，不去重
intersection	rdd1.intersection(rdd2)	{3}	返回两个 RDD 的交集，去重

表 9-2 所示是一些 Action 操作的例子。

表 9-2　一些 Action 操作的例子

函数	例子	结果	说明
collect	rdd.collect()	{1,2,3,3}	将一个 RDD 转换成数组
count	rdd.count()	4	返回 RDD 中的元素个数
take	rdd.take(2)	{1,2}	返回从 0 到 num-1 下标的 RDD 元素，不排序
top	rdd.top(2)	{3,3}	从 RDD 中，按照排序（默认为降序）返回前 num 个元素
reduce	rdd.reduce((x,y)=>x+y)	9	根据映射函数 f=x+y，对 RDD 元素进行二元计算，返回计算结果
foreach	rdd.foreach(prin)	{1,2,3,3}	遍历 RDD 每个元素，并执行 func

下面我们来看一个完整的例子。假定数据存储在 HDFS 上，而数据格式以“；”作为每列数据的分割：

```
"age";"job";"marital";"education";"default";"balance";"housing";"loan"
30;"unemployed";"married";"primary";"no";1787;"no";"no"
33;"services";"married";"secondary";"no";4789;"yes";"yes"
…
```

Scala 代码如下：

```
//1.定义了以一个 HDFS 文件（由数行文本组成）为基础的 RDD
 val lines = sc.textFile("/data/spark/bank/bank.csv")
 //2.因为首行是文件的标题，首先把首行去掉，返回新 RDD 是 withoutTitleLines
 val withoutTitleLines = lines.filter(!_.contains("age"))
 //3.将每行数据以;分割下，返回名字是 lineOfData 的新 RDD
 val lineOfData = withoutTitleLines.map(_.split(";"))
//4.获取大于30岁的数据，返回新 RDD 是 gtThirtyYearsData
 val gtThirtyYearsData = lineOfData.filter(line => line(0).toInt > 30)
 //到此，集群上还没有工作被执行。但是，用户现在已经可以在动作(action)中使用 RDD。
 //计算大于30岁的有多少人
 gtThirtyYearsData.count()
```

最后的返回结果就是大于 30 岁的人的个数。

9.5 Spark SQL

Spark SQL 是 Spark 内嵌的模块，是 Spark 的一个处理结构化数据的组件。在 Spark 程序中可以使用 SQL 查询语句或 DataFrame API。DataFrame 和 SQL 提供了通用的方式来连接多种数据源，支持从 Hive 表、外部数据库、结构化数据 Parquet 文件、JSON 文件上获得数据，并且可以在多种数据源之间执行 join 操作。

Spark SQL 的功能是通过 SQLContext 类来提供的，SQLContext 是一个入口类。要创建一个 SQLContext，首先需要一个 SparkContext，然后通过 SparkContext 实例化一个 SQLContext。在 Spark Shell 启动时，输出日志的最后有这么几条信息：

```
16/04/16 17:25:47 INFO repl.SparkILoop: Created spark context..
Spark context available as sc.
16/04/16 17:25:47 INFO repl.SparkILoop: Created sql context..
SQL context available as sqlContext.
```

这些信息表明 SparkContext 和 SQLContext 都已经初始化好了，可通过对应的 sc、sqlContext 变量直接进行访问。下述代码展示了如何创建一个 SQLContext 对象：

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
```

DataFrame（数据框架）提供了可以作为分布式 SQL 查询引擎的程序化抽象。使用 SQLContext 可以从现有的 RDD 或数据源创建 DataFrame。下面的示例程序是在安装目录的 examples/src/main/resources/目录下。我们打开 people.json 文件，这个文件是 Spark 提供的 JSON 格式的数据源文件，里面是一个有键值对（key:value）组成的 JSON 字符串，字符串之间用逗号隔开。下面是这个文件的一些内容：

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

Spark Shell 启动后，就可以用 Spark SQL API 执行数据分析查询。下面使用 sqlContext.read 函数从 JSON 文件导入数据源，创建一个 DataFrame（下面代码中的 df)：

```
val df = sqlContext.read.json
("file:///usr/local/spark/examples/src/main/resources/people.json")
```

输出结果为：

```
df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]
```

然后通过 DataFrame 的 Action 操作 show 来显示数据：

```
df.show()      // 输出数据源内容
```

输出结果为：

```
+--------+-----------+
|  age|  name|
+--------+-----------+
|  null| Michael|
|   30|   Andy|
|   19|  Justin |
+--------+-----------+
```

9.5.1　DataFrame

相对于 MapReduce API，Spark 的 RDD 抽象简化了开发，而 DataFrame 进一步抽象了数据集。如图 9-6 所示，DataFrame 类似于 RDBMS 的表，每列都有名称和类型。通过 DataFrame，可以对数据进行类似 SQL 的操作。DataFrame 将数据保存为行的集合，对应行中的各列都被命名，通过使用 DataFrame，可以非常方便地查询和过滤数据。

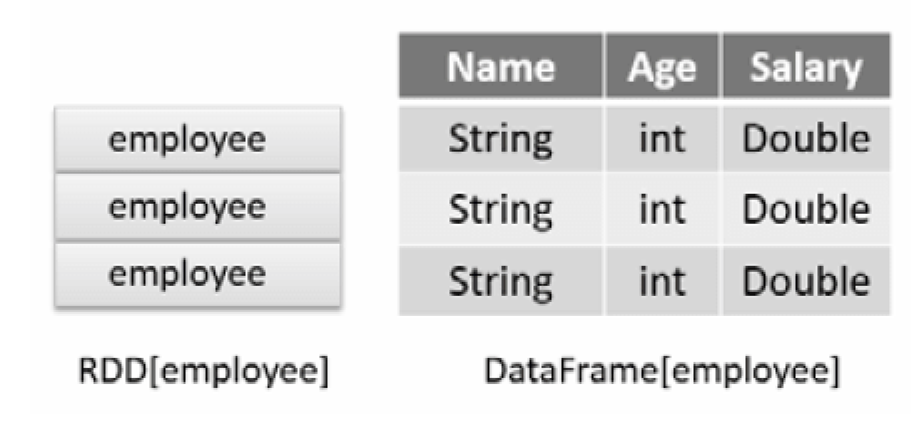


图 9-6　RDD 和 DataFrame 的比较

从体系结构上看，DataFrame 支持多种数据源。如图 9-7 所示，支持从 Hive 表、外部数据库、结构化数据 Parquet 文件、JSON 文件、RDD 等上获得数据。数据一旦被读取，借助于 DataFrames 便可以很方便地进行数据过滤、列查询、计数、求平均值及将不同数据源的数据进行整合。

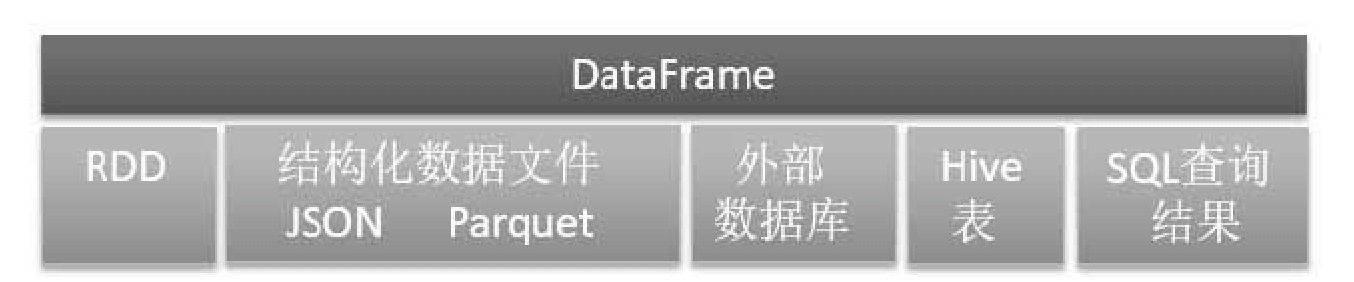


图 9-7　DataFrame 数据来源

如图 9-8 所示，使用 Spark SQL 主要分为三大步骤：

步骤 01　利用 sqlContext 从外部数据源加载数据为 DataFrame。

步骤 02　利用 DataFrame 上丰富的 API 进行查询、转换。

步骤 03　将结果进行展现或存储为各种外部数据形式。

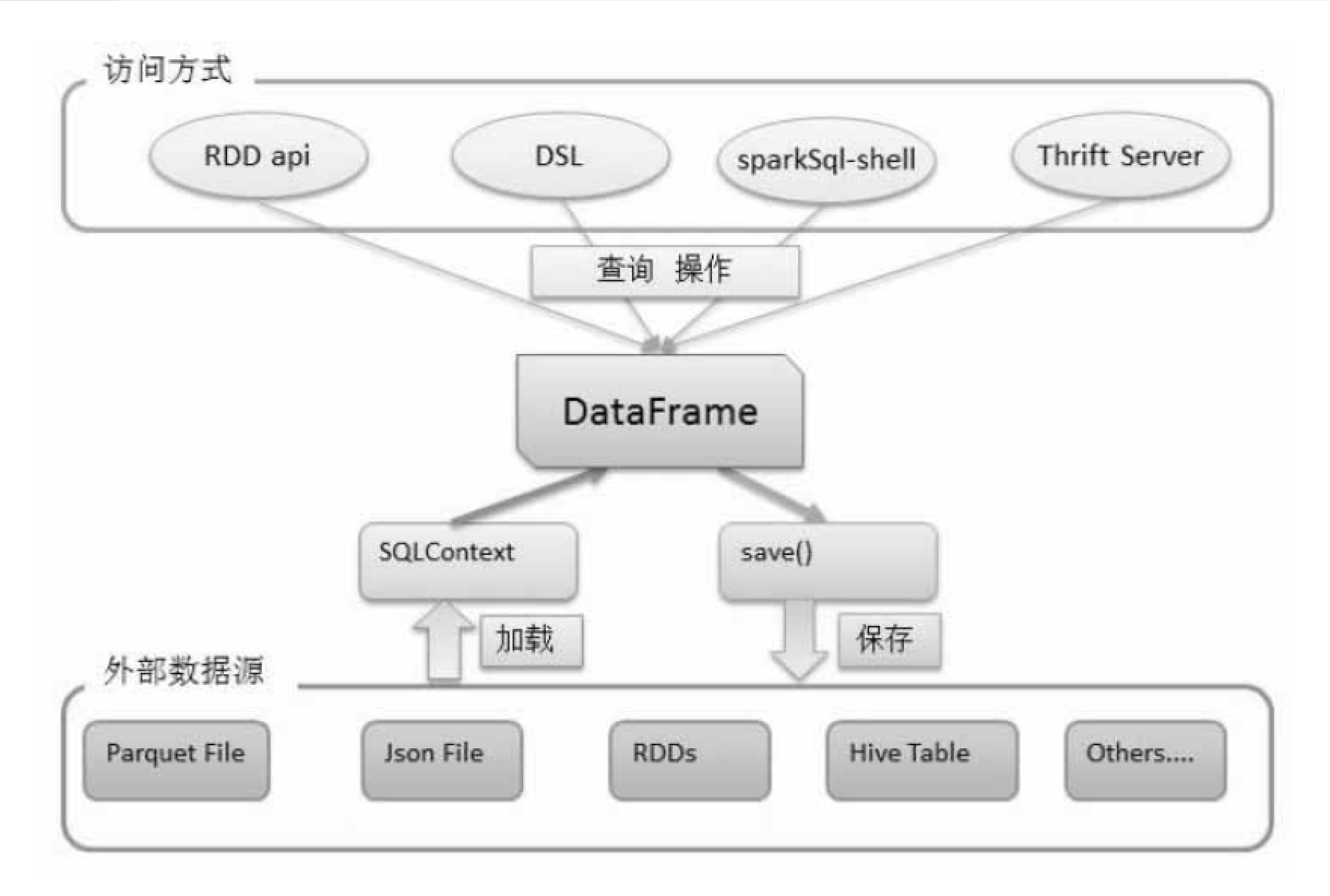


图 9-8　DataFrame

1. 加载数据

sqlContext 支持从各种各样的数据源中创建 DataFrame，内置支持的数据源有 parquetFile、JSON 文件、外部数据库、Hive 表、RDD 等：

```
# 从 Hive 中的 users 表构造 DataFrame
users = sqlContext.table("users")
# 加载 S3 上的 JSON 文件
logs = sqlContext.load("s3n://path/to/data.json", "json")
# 加载 HDFS 上的 Parquet 文件
clicks = sqlContext.load("hdfs://path/to/data.parquet", "parquet")
# 通过 JDBC 访问 MySQL
comments = sqlContext.jdbc("jdbc:mysql://localhost/comments", "user")
# 将普通 RDD 转变为 DataFrame
rdd = sparkContext.textFile("article.txt") \
               .flatMap(_.split(" ")) \
               .map((_, 1)) \
               .reduceByKey(_+_) \
wordCounts = sqlContext.createDataFrame(rdd, ["word", "count"])
# 将本地数据容器转变为 DataFrame
data = [("Alice", 21), ("Bob", 24)]
people = sqlContext.createDataFrame(data, ["name", "age"])
```

2. 使用 DataFrame

Spark DataFrame 提供了一整套用于操纵数据的 DSL。这些 DSL 在语义上与 SQL 关系查询

非常相近（这也是 Spark SQL 能够为 DataFrame 提供无缝支持的重要原因之一）。下面我们来看看 DataFrames 处理结构化数据的一些基本操作：

```
df.select("name").show()   // 只显示 "name" 列; show()是以表格形式打印结果
+----------+
|  name|
+-----------+
|Michael|
|   Andy|
|  Justin|
+-----------+

df.select(df("name"), df("age") + 1).show()   // 将 "age" 加 1
+-----------+-----------+
|   name|(age + 1)|
+-----------+------------+
| Michael|     null|
|   Andy|      31|
|   Justin|      20|
+-----------+-------------+

df.filter(df("age") > 21).show()     //过滤语句
+-----+-------+
|age|name|
+-----+-------+
|  30|Andy|
+-----+-------+

df.groupBy("age").count().show()   // groupBy 操作
+-----+--------+
| age|count|
+-----+--------+
|null|    1|
|  19|   1|
|  30|   1|
+-----+--------+
```

当然，我们也可以直接使用 SQL 语句来进行操作：

```
df.registerTempTable("people")     //将 DataFrame 注册为临时表 people
```

```
//支持在临时表上执行 SQL 查询
val result = sqlContext.sql("SELECT name, age FROM people WHERE age
>= 13 AND age <= 19")
result.show()  // 输出结果
```

输出结果为：

```
+--------+-----+
| name|age|
+--------+-----+
|Justin| 19|
+--------+----+
```

3. 保存结果

对数据的分析完成之后，可以将结果保存在多种形式的外部存储中：

```
//追加至 HDFS 上的 Parquet 文件
df.save(path="hdfs://path/to/data.parquet", source="parquet",
mode="append")
//覆写 S3上的 JSON 文件
df.save(path="s3n://path/to/data.json", source="json",mode="append")

//保存为 Hive 的内部表
df.saveAsTable(tableName="yang", source="parquet" mode="overwrite")
```

下面汇总了常见的功能：

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val df =
sqlContext.read.json("../examples/src/main/resources/people.json")
//查看所有数据：
df.show()
//查看表结构：
df.printSchema()
//只看 name 列：
df.select("name").show()
//对数据运算：
df.select(df("name"), df("age") + 1).show()
//过滤数据：
df.filter(df("age") > 21).show()
```

```
//分组统计：
df.groupBy("age").count().show()
```

更多的功能可以查看完整的 DataFrames API ，此外 DataFrames 也包含了丰富的 DataFrames Function ，可用于字符串处理、日期计算、数学计算等。

9.5.2　RDD 转化为 DataFrame

Spark SQL 支持两种不同的方法将 RDD 转化为 DataFrame。下面代码是使用反射机制来推断 RDD 的模式。Spark SQL 的 Scala 接口支持自动转换一个包含 case 类的 RDD 为一个 DataFrame。case 类定义了表的 schema，使用反射读取 case 类的参数名为列名。RDD 可以隐式转换成一个 DataFrame，并注册成一个表，表可以用于后续的 SQL 查询语句。

下面我们来看一个例子。假定一个文本文件 customers.txt 中的内容如下：

```
100, John Smith, Austin, TX, 78727
200, Joe Johnson, Dallas, TX, 75201
300, Bob Jones, Houston, TX, 77028
400, Andy Davis, San Antonio, TX, 78227
500, James Williams, Austin, TX, 78727
…
```

我们从文本文件中加载用户数据，并从数据集中创建一个 DataFrame 对象。然后运行 DataFrame 函数，执行特定的数据选择查询。下述的代码也可以在 Spark Shell 终端执行的 Spark SQL 命令。

```
// 首先用已有的 Spark Context 对象创建 SQLContext 对象
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
// 导入语句，可以隐式地将 RDD 转化成 DataFrame
import sqlContext.implicits._
// 创建一个表示客户的自定义类
case class Customer(customer_id: int, name: String, city: String,
state: String, zip_code: String)
// 用数据集文本文件创建一个 Customer 对象的 DataFrame。读取文件创建一个
//MappedRDD，并将数据写入 Customer 模式类，隐式转换为 DataFrame
val dfCustomers =
sc.textFile("data/customers.txt").map(_.split(",")).map(p =>
Customer(p(0).trim.toInt, p(1), p(2), p(3), p(4))).toDF()
// 显示 DataFrame 的内容
dfCustomers.show()
// 打印 DF 模式
```

```
    dfCustomers.printSchema()
    // 选择客户名称列
    dfCustomers.select("name").show()
    // 选择客户名称和城市列
    dfCustomers.select("name", "city").show()
    // 根据 id 选择客户
    dfCustomers.filter(dfCustomers("customer_id").equalTo(500)).show()
    // 根据邮政编码统计客户数量
    dfCustomers.groupBy("zip_code").count().show()

    // 将 DataFrame 注册为一个表，供查询使用
    dfCustomers.registerTempTable("customers")
    //使用 SQL
    val  result  =sqlContext.sql("SELECT  name,city  FROM  customers  where
customer_id<400")
    //SQL 查询结果是 DataFrame，可通过索引和字段名访问
```

在上一示例中，模式是通过反射而得来的。我们也可以通过编程的方式指定数据集的模式。这种方法在由于数据的结构以字符串的形式编码而无法提前定义定制类的情况下非常实用。下面这个例子展示了如何使用数据类型类 StructType、StringType 和 StructField 指定模式。

```
    // 用编程的方式指定模式
    // 用已有的 Spark Context 对象创建 SQLContext 对象
    val sqlContext = new org.apache.spark.sql.SQLContext(sc)
    // 创建 RDD 对象
    val rddCustomers = sc.textFile("data/customers.txt")
    // 用字符串编码模式
    val schemaString = "customer_id name city state zip_code"
    // 导入 Spark SQL 数据类型和 Row
    import org.apache.spark.sql._
    import org.apache.spark.sql.types._;
    // 用模式字符串生成模式对象
    val  schema  =  StructType(schemaString.split("  ").map(fieldName  =>
StructField(fieldName, StringType, true)))
    // 将 RDD（rddCustomers）记录转化成 Row
    val rowRDD = rddCustomers.map(_.split(",")).map(p =>
Row(p(0).trim,p(1),p(2),p(3),p(4)))
    // 将模式应用于 RDD 对象。
    val dfCustomers = sqlContext.createDataFrame(rowRDD, schema)
    // 将 DataFrame 注册为表
```

```
    dfCustomers.registerTempTable("customers")
    // 用 sqlContext 对象提供的 sql 方法执行 SQL 语句。
    val custNames = sqlContext.sql("SELECT name FROM customers")
    // SQL 查询的返回结果为 DataFrame 对象，支持所有通用的 RDD 操作。
    // 可以按照顺序访问结果行的各个列。
    custNames.map(t => "Name: " + t(0)).collect().foreach(println)
    // 用 sqlContext 对象提供的 sql 方法执行 SQL 语句。
    val   customersByCity   =   sqlContext.sql("SELECT   name,zip_code   FROM
customers ORDER BY zip_code")
    // SQL 查询的返回结果为 DataFrame 对象，支持所有通用的 RDD 操作。
    // 可以按照顺序访问结果行的各个列。
    customersByCity.map(t => t(0) + "," + t(1)).collect().foreach(println)
```

Spark SQL 的调优参数可以用来性能调优。比如：

- spark.sql.inMemoryColumnarStorage.compressed：默认值为 true。当设置为 true 时，Spark SQL 将为基于数据统计信息为每列自动选择一个压缩算法。
- spark.sql.inMemoryColumnarStorage.batchSize：默认为 10000，控制列式缓存的批处理大小。大批量可以提高内存的利用率以及压缩率，但有 OOM 的风险。

9.5.3　JDBC 数据源

在 9.5.1 小节中阐述了从一个 JSON 数据源中加载数据为 DataFrame，Spark SQL 还支持 JDBC 数据源。JDBC 数据源可用于通过 JDBC API 读取关系型数据库中的数据。JDBC 数据源能够将结果作为 DataFrame 对象返回，并直接用 Spark SQL 处理或 join 其他数据源。

为了访问某一个关系数据库，需要将其驱动添加到 classpath，例如：

```
    SPARK_CLASSPATH=postgresql-9.3-1102-jdbc41.jar bin/spark-shell
```

访问 JDBC 数据源需要提供以下参数：

- url：待连接的 JDBC URL;
- dbtable：被读的 JDBC 表;
- driver：JDBC 驱动程序的类名;
- partitionColumn、lowerBound、upperBound、numPartitions：指定了多个 workers 并行读数据时如何分区表。

Scala 的代码为：

```
    val jdbcDF = sqlContext.load("jdbc", Map( "url" ->
"jdbc:postgresql:dbserver",
```

```
    "dbtable" -> "schema.tablename"))
```

Java 的代码为：

```
Map<String, String> options = new HashMap<String, String>();
options.put("url", "jdbc:postgresql:dbserver");
options.put("dbtable", "schema.tablename");
DataFrame jdbcDF = sqlContext.load("jdbc", options)
```

9.5.4 Hive 数据源

Spark SQL 支持从 Hive 表中读写数据。Spark SQL 支持的功能有：

（1）查询语句：SELECT、GROUP BY、ORDER BY、CLUSTER BY、SORT BY；

（2）Hive 操作运算：

- 关系运算：= 、==、<>、<、>、>=、<=等。
- 算术运算：+、-、*、/、%等。
- 逻辑运算：AND、&&、OR、||等。
- 数学函数：(sign、ln、cos 等)。
- 字符串函数：instr、length、printf 等。

（3）用户自定义函数（UDF）；

（4）用户自定义聚合函数（UDAF）；

（5）用户定义的序列化格式（SerDes）；

（6）join 操作：JOIN、{LEFT|RIGHT|FULL} OUTER JOIN、LEFT SEMI JOIN、CROSS JOIN；

（7）unions 操作；

（8）子查询：SELECT col FROM (SELECT a + b AS col from t1) t2；

（9）抽样（Sampling）；

（10）解释（Explain）；

（11）分区表；

（12）Hive DDL 函数：CREATE TABLE、CREATE TABLE AS SELECT、ALTER TABLE；

（13）Hive 数据类型：TINYINT、SMALLINT、INT、BIGINT、BOOLEAN、FLOAT、DOUBLE、STRING、BINARY、TIMESTAMP、DATE、ARRAY、MAP、STRUCT。

Spark SQL 中的 HiveContext 通过基本的 SQLContext 提供了一系列的方法集，可以使用 HiveQL 解析器编写查询语句以及从 Hive 表中读取数据时使用。比如：

Scala 代码：

```
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)
```

```
    sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
    sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")
    // Queries are expressed in HiveQL
    sqlContext.sql("FROM src SELECT key, value").collect().foreach(println)
```

Java 代码：

```
    // sc is an existing JavaSparkContext.
    HiveContext sqlContext = new org.apache.spark.sql.hive.HiveContext(sc);
    sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)");
    sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src");
    // Queries are expressed in HiveQL.
    Row[] results = sqlContext.sql("FROM src SELECT key, value").collect();
```

9.6 Spark Streaming

流计算除了使用 Storm 框架，使用 Spark Streaming 也是一个很好的选择。虽然 Storm 保证了实时性，但实现的复杂度大大提高。而 Spark Streaming 能够以准实时的方式相对容易地实现较为复杂的数据处理。基于 Spark Streaming，可以方便地构建可拓展、高容错的流计算应用程序。

Spark Streaming 是构建在 Spark 上处理 Stream 数据的框架。如图 9-9 所示，它并不会像 Storm 那样一次一个地处理数据流，而是在处理前按时间间隔预先将 Stream 数据切分为一段一段的批处理作业（时间片断为几秒），以类似批量处理的方式来处理这些切分好的数据。

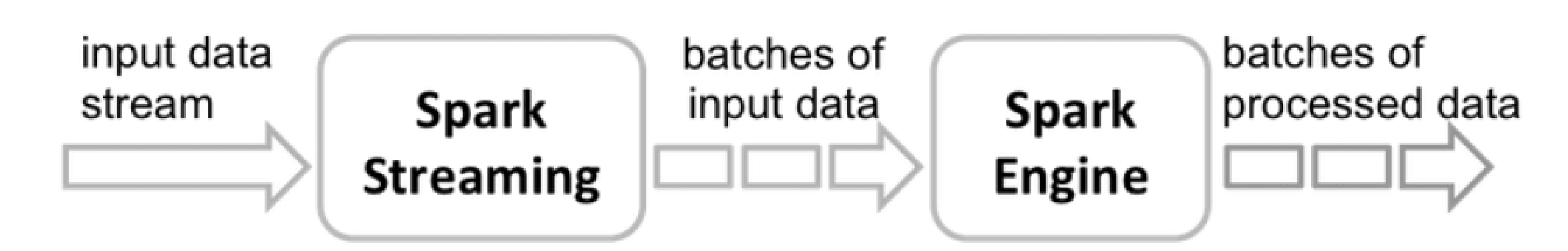


图 9-9 Spark Streaming 工作原理

Spark Streaming 在接收到实时数据后，给数据分批次，然后传给 Spark Engine 处理，最后生

成该批次的结果。它支持的持续性数据流叫 DStream（Discretized Stream），直接支持 Kafka、Flume 的数据源。DStream 是一种连续的 RDD。

Spark Streaming 构建在 Spark 上，一方面是因为 Spark 的低延迟执行引擎（100ms+）可以用于实时计算，另一方面相比 Storm，Spark 的 RDD 数据集更容易做高效的容错处理。此外，小批量处理的方式使得它可以同时兼容批量和实时数据处理的逻辑和算法。方便了一些需要历史数据和实时数据联合分析的特定应用场合。Spark Streaming 使用 Spark API 进行流计算，在 Spark 上进行流处理与批处理的方式一样。因此，你可以复用批处理的代码，使用 Spark Streaming 构建强大的交互式应用程序，而不仅仅是用于分析数据。

如图 9-10 所示，Spark Streaming 可以接受来自 Kafka、Flume 和 TCP Socket 等的数据源，使用简单的 API 函数比如 map、reduce、join 等操作，就可以直接使用内置的机器学习算法和图算法包来处理数据。经过处理的结果可以存储在文件系统（如：HDFS）、数据库（如：HBase）等存储系统上。



图 9-10　Spark Streaming 输入输出示意图

9.6.1　DStream 编程模型

下面这个例子帮助大家理解 DStream。这个例子是基于流的单词统计：本地服务器通过 TCP 接收文本数据，实时输出单词统计结果。运行该示例需要 Netcat（在网络上通过 TCP 或 UDP 读写数据），CentOS 6.x 系统中默认没有安装。我们选择 Netcat 0.6.1 版本，在终端中运行如下命令进行下载并安装：

```
   wget
http://downloads.sourceforge.net/project/netcat/netcat/0.6.1/netcat-
0.6.1-1.i386.rpm -O ~/netcat-0.6.1-1.i386.rpm
   sudo rpm -iUv ~/netcat-0.6.1-1.i386.rpm  #安装
```

安装好 Netcat 之后，使用如下命令建立本地数据服务，监听 TCP 端口 9999：

```
   #终端 1
   nc -l -p 9999
```

启动后，该端口就被占用了，需要开启另一个终端，并运行示例程序，执行如下命令：

```
   #终端 2
```

```
/usr/local/spark/bin/run-example streaming.NetworkWordCount localhost 9999
```

接着在终端 1 中输入文本，在终端 2 中就可以实时看到单词统计结果了。最后需要关掉终端 2，并按 Ctrl+C 组合键退出终端 1 的 Netcat。

对于上面的基于流的单词统计的例子，下面是其一部分代码：

```
//首先实例化一个 StreamingContext，批次间隔为1秒
val ssc = new StreamingContext(sparkConf, Seconds(1));
//调用 StreamingContext 的 socketTextStream，创建一个 DStream，连接到端口
val lines = ssc.socketTextStream(serverIP, serverPort);
// 对获得的 DStream 进行处理，将每一行数据执行 Split 操作，切分成单词
val words = lines.flatMap(_.split(" "));
// 将每个单词转换为（单词，1）的形式，形成 MappedDStream。
//下面的操作类似 RDD 的转换操作
val pairs = words.map(word => (word, 1));
//使用 reduceByKey 将相同单词的值计数出来，统计 word 的数量
val wordCounts = pairs.reduceByKey(_ + _);
// 输出结果，也可保存到外部系统上
wordCounts.print();
ssc.start();              // 开始
ssc.awaitTermination();  // 计算完毕退出
```

我们再看 StreamingContext 的 socketTextStream 代码：

```
def socketTextStream(
        hostname: String,
        port: Int,
        storageLevel: StorageLevel = StorageLevel.MEMORY_AND_DISK_SER_2
      ): ReceiverInputDStream[String] = {
      socketStream[String](hostname, port, SocketReceiver.bytesToLines, storageLevel)
}
```

上述代码使用 SocketReceiver 的 bytesToLines 把输入流转换成可遍历的数据。如果我们继续看 socketStream，那么，它是 new 了一个 SocketInputDStream 对象。如果查看它的继承关系，则 SocketInputDStream >> ReceiverInputDStream >> InputDStream >> DStream。因此，DStream 是高级抽象连续数据流，一个 DStream 可以看作是一个 RDD 的序列。

下面这个代码完成了上述的功能：

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.streaming.Durations;
import org.apache.spark.streaming.api.java.JavaDStream;
import org.apache.spark.streaming.api.java.JavaPairDStream;
import org.apache.spark.streaming.api.java.JavaReceiverInputDStream;
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import scala.Tuple2;

import java.util.Arrays;

public class SparkStreamingDemo {
    public static void main(String[] args) {
        SparkConf conf =
 new SparkConf().setMaster("local[2]").setAppName("NetworkWordCount");
        JavaStreamingContext jssc = new JavaStreamingContext(conf,
Durations.seconds(10));
        JavaReceiverInputDStream<String> lines =
jssc.socketTextStream("localhost", 9999);

        // Split each line into words
        JavaDStream<String> words = lines.flatMap(
                new FlatMapFunction<String, String>() {
                    public Iterable<String> call(String x) {
                        return Arrays.asList(x.split(" "));
                    }
                });

        // Count each word in each batch
        JavaPairDStream<String, Integer> pairs = words.mapToPair(
                new PairFunction<String, String, Integer>() {
                    public Tuple2<String, Integer> call(String s) {
                        return new Tuple2<String, Integer>(s, 1);
                    }
                });
        JavaPairDStream<String,        Integer>        wordCounts        =
pairs.reduceByKey(
```

```
                new Function2<Integer, Integer, Integer>() {
                    public Integer call(Integer i1, Integer i2) {
                        return i1 + i2;
                    }
                });

        // Print the first ten elements of each RDD generated in this
DStream to the console
        wordCounts.print();

        jssc.start();              // Start the computation
        jssc.awaitTermination();     // Wait for the computation to
terminate
    }
}
```

9.6.2　DStream 操作

类似 RDD，对于 DStream，我们可以进行多种操作：transformations（转换）、状态、output（输出）等。

常见的转换操作有：

- map(func)：对每一个元素执行 func 函数，返回一个新的 DStream;
- flatMap(func)：类似 map 函数，但是可以 map 到 0+个输出;
- filter(func)：过滤，返回一个新的 DStream;
- repartition(numPartitions)：通过增加分区，提高 DStream 的并行度;
- union(otherStream)：合并两个 DStream 的元素为一个新的 DStream;
- count()：统计 DStream 中每个 RDD 元素的个数;
- reduce(func)：对 DStream 中的每个 RDD 进行聚合操作（2 个输入参数，1 个输出参数）;
- countByValue()：针对类型统计，当一个 DStream 的元素的类型是 K 的时候，调用它会返回一个新的 DStream，包含<K,Long>键值对，Long 是每个 K 出现的频率;
- reduceByKey(func, [numTasks])：对于一个(K, V)类型的 DStream，为每个 key，执行 func 函数；默认 local 是 2 个线程，cluster 是 8 个线程，也可以指定 numTasks;
- join(otherStream, [numTasks])：把(K, V)和(K, W)的 DStream 连接成一个(K, (V, W))的新 DStream;
- cogroup(otherStream, [numTasks])：把(K, V)和(K, W)的 DStream 连接成一个(K, Seq[V], Seq[W])的新 DStream;
- transform(func)：转换操作，把原来的 RDD 通过 func 转换成一个新的 RDD。

常见的状态操作有：

- updateStateByKey(func)：针对 key 使用 func 来更新状态和值，返回一个新状态的 DStream，该状态可以为任何值。使用这个操作，我们希望保存它状态的信息，然后持续地更新它，使用它有两个步骤：（1）定义状态，这个状态可以是任意的数据类型；（2）定义状态更新函数，从前一个状态更改新的状态；
- reduceByKeyAndWindow(func…)：Windows 操作，具体见下面的解释。

对于 Windows 操作，我们先看个例子。比如前面的 word count 的例子，我们想要每隔 10 秒计算一下最近 30 秒的单词总数。那么，我们可以使用以下语句：

```
    // Reduce last 30 seconds of data, every 10 seconds
    val windowedWordCounts = pairs.reduceByKeyAndWindow(_ + _,
Seconds(30), Seconds(10))
```

上面用到了 Windows 的两个参数：

- window length：window 的长度是 30 秒，最近 30 秒的数据；
- slice interval：计算的时间间隔。

窗口的作用之一是定期计算滑动的数据。如图 9-11 所示，这个图来自 Spark 官网，上面的部分是 DStream，下面的部分是 Windows 计算后的 DStream。

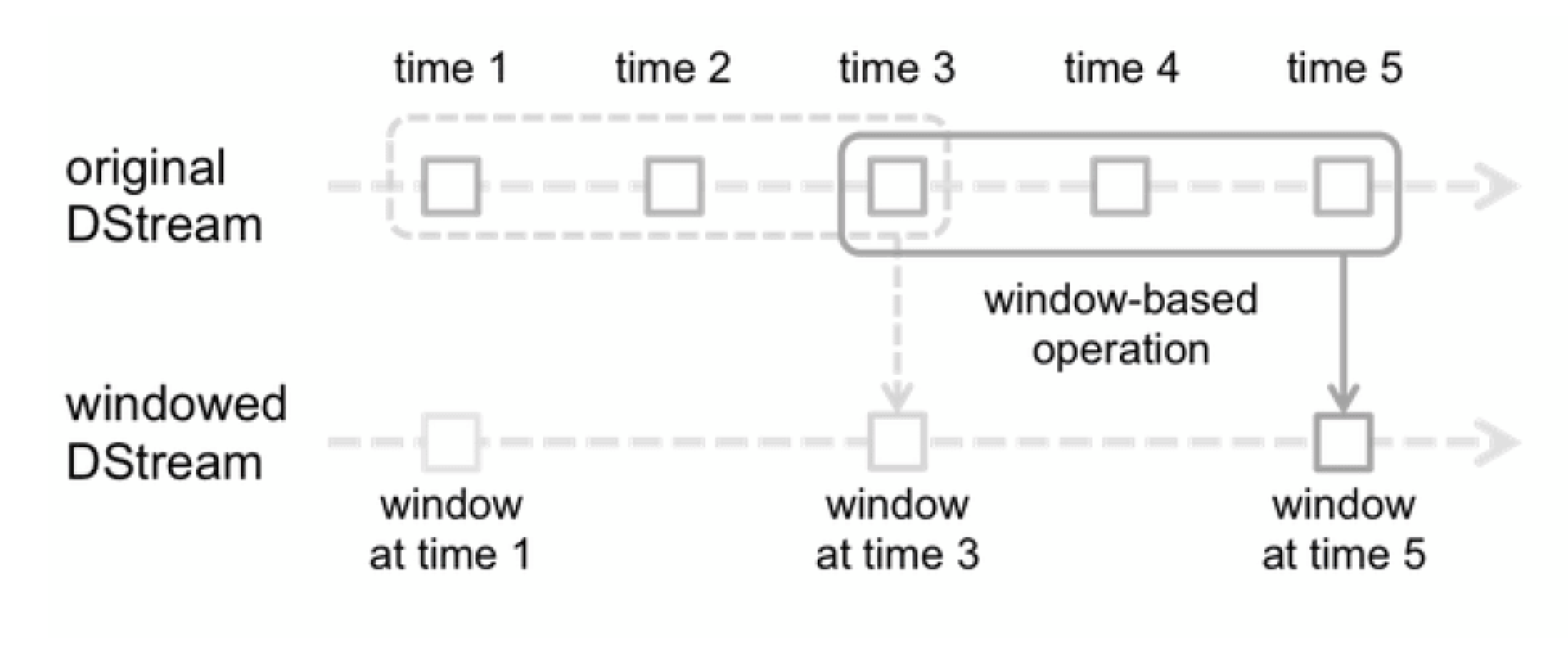


图 9-11　滑动窗口计算

常见的输出操作为：

- print()：打印到控制台；
- foreachRDD(func)：对 DStream 里面的每个 RDD 执行 func，保存到外部系统；
- saveAsObjectFiles(prefix,[suffix])：保存流的内容为 SequenceFile，文件名："prefix-TIME_IN_MS[.suffix]";
- saveAsTextFiles(prefix,[suffix])：保存流的内容为文本文件，文件名："prefix-TIME_IN_MS[.suffix]";

- saveAsHadoopFiles(prefix, [suffix])：保存流的内容为 hadoop 文件,文件名 ： "prefix-TIME_IN_MS[.suffix]"。

Spark Streaming 提供了检查点的功能，实现容错机制。我们知道，状态的操作是基于多个批次的数据。它包括基于 Windows 的操作和 updateStateByKey。因为状态的操作要依赖于上一个批次的数据，所以它要根据时间，不断累积元数据。为了清空数据，它支持周期性的检查点，通过把中间结果保存到 HDFS 上。因为检查操作会导致保存到 HDFS 上的开销，所以设置这个时间间隔需要非常慎重。对于小批次的数据，比如一秒的，检查操作会大大降低吞吐量。但是检查的间隔太长，会导致任务变大。通常来说，5~10 秒的检查间隔时间是比较合适的。

```
ssc.checkpoint(hdfsPath)  //设置检查点的保存位置
dstream.checkpoint(checkpointInterval)  //设置检查点间隔
```

对于必须设置检查点的 DStream，比如通过 updateStateByKey 和 reduceByKeyAndWindow 创建的 DStream，默认设置是至少 10 秒。

9.6.3　性能考虑

对于调优，可以从两个方面考虑：

（1）利用集群资源，减少处理每个批次的数据的时间。
（2）给每个批次的数据量设定一个合适的大小。

像一些分布式的数据处理操作，比如 reduceByKey 和 reduceByKeyAndWindow，默认为 8 个并发线程。我们可以提高数据处理的并行度。通过修改参数 spark.default.parallelism 来提高这个默认值。对于接收数据的任务，也可以提高其接收的并行度。

为了使流处理能在集群上稳定地运行，要使处理数据的速度跟上数据流入的速度。最好的方式是计算这个批量的大小，我们首先设置 batch size 为 5~10 秒和一个很低的数据输入速度。确定系统能跟上数据的输入速度的时候，我们可以根据经验设置批次的大小，通过查看日志获得 Total delay（总延迟）为多长时间。如果 delay（延迟时间）小于 batch（批处理时间），那么系统是稳定的；如果 delay 一直增加，说明系统的处理速度跟不上数据的输入速度。

优化内存使用是非常重要的。DStream 默认的持久化级别是 MEMORY_ONLY_SER，而不是 RDD 的 MEMORY_ONLY。Streaming 会将接收到的数据全部存储于可用的内存区域内，因此对于已经完成处理的数据应该及时清理，以确保 Streaming 有足够的内存。默认地，所有 Spark Streaming 生成的持久化 RDD 都会通过 LRU 算法清除出内存。通过设置 spark.cleaner.ttl，Streaming 就能自动地定期清除旧的内容。但是设置这个参数要很谨慎。另一个方法是设置 spark.streaming.unpersist 为 true 来启用内存清理，减少 RDD 内存的使用，这样有利于提升 GC 的性能。推荐使用并行 mark-and-sweep GC 来减少 GC 的突然暂停的情况，虽然这样会降低系统的吞吐量，但是这样有助于系统更稳定地进行批处理。

9.6.4 容错能力

如果全部输入数据是在 HDFS 上，因为 HDFS 是可靠的文件系统，所以不会有任何的数据失效。如果数据来源是网络，比如 Kafka 和 Flume，为了防止失效，默认是数据会保存到 2 个节点的内存中，但是有一种可能性是接收数据的节点挂了，那么数据可能会丢失，因为它还没来得及把数据复制到另外一个节点。

为了支持 24×7 不间断的处理，Spark 支持驱动节点失效后，重新恢复计算。Spark Streaming 会周期性地写数据到 HDFS 系统，就是前面的检查点的那个目录。驱动节点失效之后，StreamingContext 可以被恢复的。为了让一个 Spark Streaming 程序能够被恢复，它需要做以下操作：

（1）第一次启动的时候，创建 StreamingContext，创建所有的 streams，然后调用 start()方法。

（2）恢复时，必须通过检查点的数据重新创建 StreamingContext。

下面是一个设置检查点的例子：

```
def functionToCreateContext(): StreamingContext = {
    val ssc = new StreamingContext(...)   // new context
    val lines = ssc.socketTextStream(...)   // create DStreams
    …
    ssc.checkpoint(checkpointDirectory)  // 设置检查点目录
    …
}
```

9.7 GraphX 图计算框架

图论是研究一组实体（称为顶点）之间两两关系（称为边）的特点。图论对商业领域产生了深远的影响。几乎所有大型互联网公司都建立了关系网络，通过对这些重要的关系网络进行分析，这些公司获得了巨大的价值。比如：亚马逊和 Netfilx 分别建立并掌握了顾客-商品购买关系和用户-电影评分关系，并且基于这些关系构建各自的推荐算法。Facebook 和 LinkedIn 则构建了人类关系图谱并对这些关系进行分析。在构建关系网络方面，最有名的例子是谷歌的 PageRank 算法，根据互联网上的链接来源来判断网页的重要程度。随着图谱越来越大，不断有新的并行图处理框架被开发出来，比如：谷歌的 Pregel、雅虎的 Giraph 和卡内基梅隆大学的 GraphLab 等。这些框架以图处理为中心，支持容错和迭代式内存计算，大大提高了图计算的处理效率。

Spark 的 GraphX 是一个分布式图处理框架，支持 Pregel、Giraph 和 GraphLab 中的许多图并行处理任务。有了 GraphX，你能够使用熟悉的 Spark 抽象来进行图并行编程。Spark GraphX 基于 Spark 平台提供对图计算和图挖掘简洁易用的接口，极大地方便了大家对分布式图处理的需

求。GraphX 用两个 RDD：顶点 RDD（VertexRDD）和边 RDD（EdgeRDD）来表示图。GraphX 的 API 类似谷歌的 Pregel。

大家都知道，社交网络中人与人之间有很多关系链，例如 Twitter、Facebook、微博、微信，这些都是大数据产生的地方，都需要分布式图计算，而并非单机处理，Spark GraphX 由于底层是基于 Spark 来处理的，所以天然就是一个分布式的图处理系统。图的分布式或者并行处理其实是把一张大图拆分成很多的子图，然后我们分别对这些子图进行计算，计算的时候可以分别迭代进行分阶段的计算，即对图进行并行计算。

Spark GraphX 的优势在于能够把表格和图进行互相转换。通过抽象为弹性分布式属性图（resilient distributed property graph），GraphX 统一了 Table 和 Graph 两种视图。只需要一份物理存储，就可以拥有表和图的两种表示形式，而这两种表现形式都有自己的操作符，从而使得操作非常灵活和高效。图 9-12 形象地表示了同一原始数据可能有许多不同表和图的视图。在获得原始的 Wikipedia 的文档以后，可以变成 Table 形式的视图，然后基于 Table 形式的视图分析 Hyperlinks 超链接，也可以分析 Term-Doc Graph，然后经过 LDA 之后进入 WordTopics；对于 Hyperlinks，我们可以使用 PageRank 去分析。在下面的 Editor Graph 到 Community，这个过程可以称之为 Triangle Computation，这是计算三角形的一个算法，基于此就会发现一个社区。从上面的分析中，我们可以发现图计算有很多的做法和算法，同时也发现图和表格可以做互相的转换。

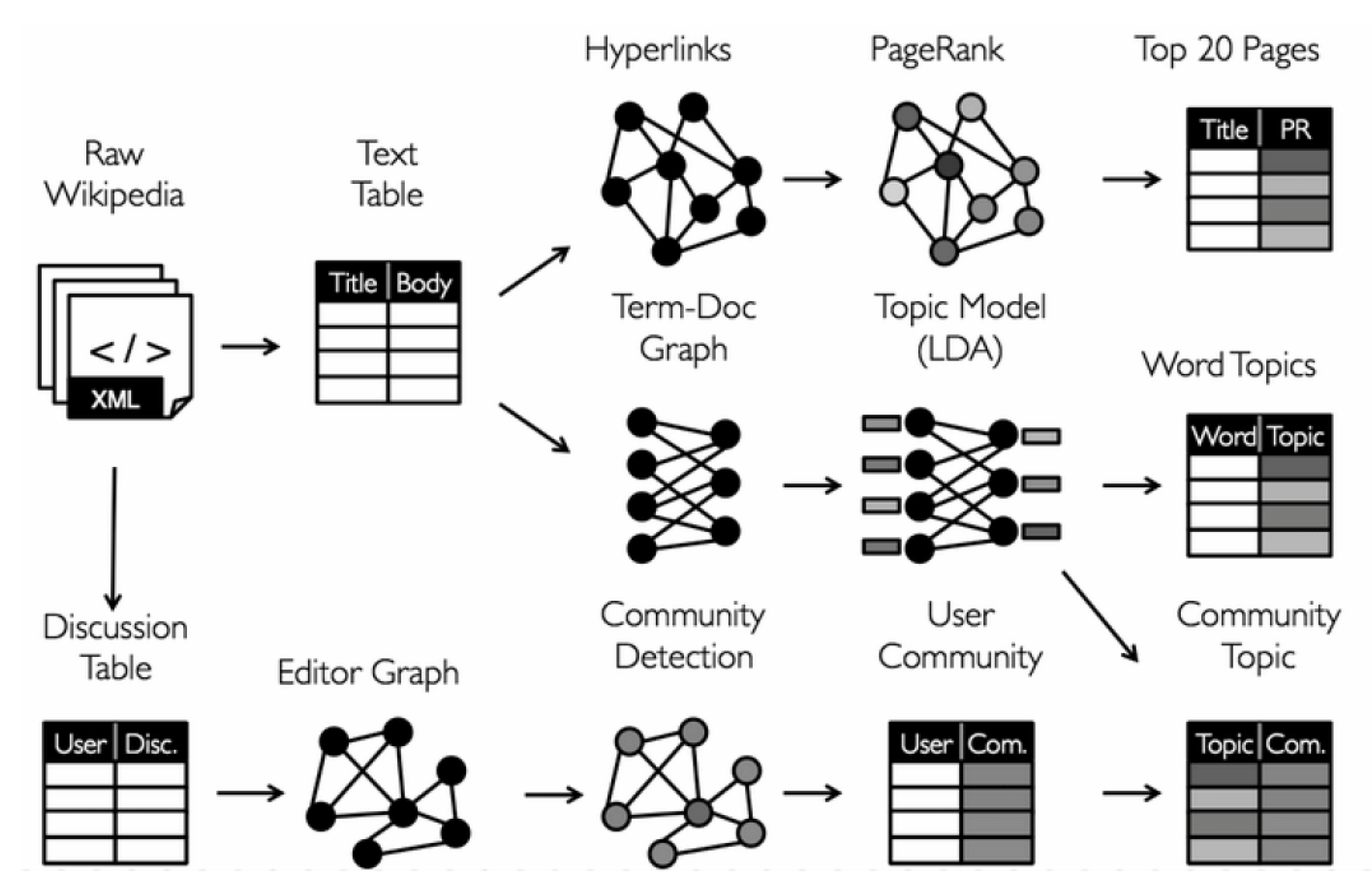
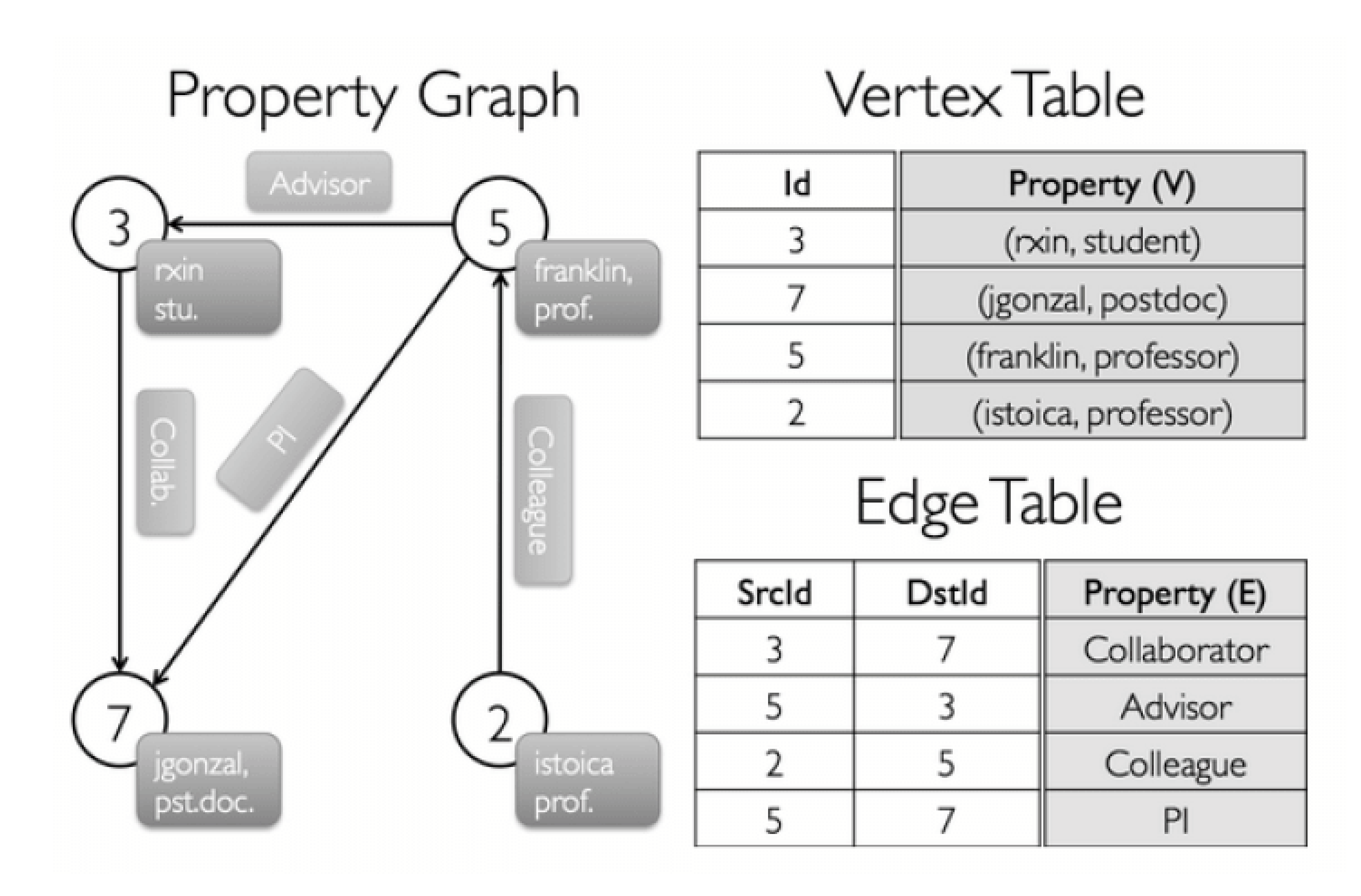


图 9-12　图和表的视图切换

9.7.1 属性图

在 Spark GraphX 中的 Graph 其实是 Property Graph（属性图），也就是说图的每个顶点和边都是有属性的，如图 9-13 所示。



Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

图 9-13　属性图

在图 9-13 中，构造了一个包括不同人员（用户）的属性图。顶点属性包含用户名和职业，人员之间关系是用字符串标注。顶点 3 的名称为 rxin，是学生；顶点 5 是 franlin，是一个教授，5 到 3 的边是表明 5 是 3 的 Advisor（导师），上图中灰色的表示的是相应顶点的 Property，而浅灰色部分表示边的 Property，边和顶点都是有 ID 的，对于顶点而言有自身的 ID，而对于边来说有 SourceID（起点）和 DestinationID（终点），即对于边而言会有两个 ID 来表达从哪个顶点出发到哪个顶点结束，同时也表明了边的方向，这就是 Property Graph 的表示方法。

例如，在 Vertex Table 中 ID 为的 3 的 Property 就是(rxin, student)，而在 Edge Table 中 3 到 7 的边的 Property 是 Collaborator 的关系，2 到 5 是 Colleague 的关系。更为重要的是 Property Graph 和 Table 之间是可以相互转换的，在 GraphX 中所有操作的基础是 table operator 和 graph operator,，其继承自 Spark 中的 RDD，都是针对集合进行操作。

GraphX 上的图是带有顶点和边属性的有向多重图，并且扩展了 Spark RDD。顶点 Vertices 对应的 RDD 名称为 VertexRDD， VertexRDD 继承自 RDD[(VertexId, VD)]，RDD 的类型是 VertexId 和 VD，其中的 VD 是顶点属性的类型，也就是说 VertexRDD 有 ID 和顶点属性。Edges 对应的是 EdgeRDD，属性有三个：源顶点的 ID、目标顶点的 ID、边属性。EdgeRDD 继承自 RDD[Edge[ED]]。Graph 类就是包含该图的顶点和边的类：

```
class Graph[VD, ED] {
 val vertices: VertexRDD[VD]
 val edges: EdgeRDD[ED]
}
```

针对图 9-13 的例子，我们有如下的代码来构建整个图：

```
// 假定 SparkContext 已经构建
val sc: SparkContext
// 为顶点创建 RDD
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal",
"postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))

// 为边创建 RDD
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L,
"advisor"),                Edge(2L, 5L, "colleague"), Edge(5L, 7L,
"pi")))

// Define a default user in case there are relationship with missing
user
val defaultUser = ("John Doe", "Missing")
// 构建初始的 Graph
val graph = Graph(users, relationships, defaultUser)
```

在上面的例子中，我们用到了 Edge case 类。边有一个 srcId 和 dstId 分别对应于源和目标顶点的标示符。另外，Edge 类有一个 attr 成员用来存储边属性。我们可以分别用 graph.vertices 和 graph.edges 成员将一个图解析为相应的顶点和边。

```
val graph: Graph[(String, String), String] // Constructed from above
// 所有 postdocs 的人数
graph.vertices.filter { case (id, (name, pos)) => pos ==
"postdoc" }.count
// 所有 src > dst 的边的个数
graph.edges.filter(e => e.srcId > e.dstId).count
```

在上面这个例子中，graph.vertices 返回一个 VertexRDD[(String, String)]，它继承于 RDD[(VertexID, (String, String))]。所以我们可以用 Scala 的 case 表达式解析这个元组。另一方面，graph.edges 返回一个包含 Edge[String]对象的 EdgeRDD。我们也可以用到 case 类的类型构造器：

```
graph.edges.filter { case Edge(src, dst, prop) => src > dst }.count
```

除了属性图的顶点和边视图，GraphX 也包含了一个三元组视图，三元视图逻辑上将顶点和边的属性保存为一个 RDD[EdgeTriplet[VD, ED]]，它包含 EdgeTriplet 类的实例。可以通过下面的

SQL 表达式表示这个连接。

```
SELECT src.id, dst.id, src.attr, e.attr, dst.attr
FROM edges AS e LEFT JOIN vertices AS src, vertices AS dst
ON e.srcId = src.Id AND e.dstId = dst.Id
```

图 9-14 描述了三元组视图与顶点和边的关系。

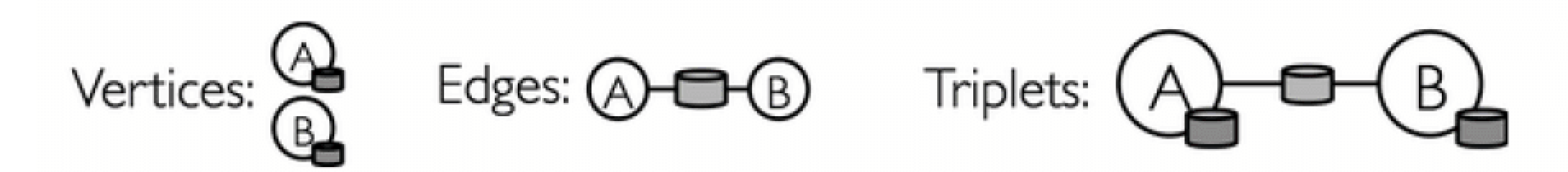


图 9-14　三元组视图

EdgeTriplet 类继承于 Edge 类，并且加入了 srcAttr 和 dstAttr 成员，这两个成员分别包含源和目的的属性。我们可以用一个三元组视图渲染字符串集合来描述用户之间的关系。

```
val graph: Graph[(String, String), String] // Constructed from above
// Use the triplets view to create an RDD of facts.
val facts: RDD[String] =
  graph.triplets.map(triplet =>
    triplet.srcAttr._1 + " is the " + triplet.attr + " of " +
triplet.dstAttr._1)
facts.collect.foreach(println(_))
```

最后还有一点，属性图通过 vertex(VD)和 edge(ED)类型参数化，这些类型是分别与每个顶点和边相关联的对象的类型。在某些情况下，在相同的图形中，可能希望顶点拥有不同的属性类型。这可以通过继承完成。例如，将用户和产品建模成一个二分图，我们可以用如下方式：

```
class VertexProperty()
case class UserProperty(val name: String) extends VertexProperty
case class ProductProperty(val name: String, val price: Double)
extends VertexProperty
// The graph might then have the type:
var graph: Graph[VertexProperty, String] = null
```

9.7.2　图操作符

正如 RDD 有基本的操作 map、filter 和 reduceByKey 一样，属性图也有基本的集合操作。核心操作是定义在 Graph 中，便捷操作定义在 GraphOps 中。然而， 因为有 Scala 的隐式转换，定义在 GraphOps 中的操作可以作为 Graph 的成员自动使用。例如，我们可以通过下面的方式计算每个顶点的入度（定义在 GraphOps 中）：

```
val graph: Graph[(String, String), String]
// Use the implicit GraphOps.inDegrees operator
val inDegrees: VertexRDD[Int] = graph.inDegrees
```

区分核心图操作和 GraphOps 的原因是为了在将来支持不同的图表示。每个图表示都必须提供核心操作的实现，并重用很多定义在 GraphOps 中的有用操作。下面汇总了 Graph 和 GraphOps 中的操作：

```
/** Summary of the functionality in the property graph */
class Graph[VD, ED] {
  //          Information          about          the          Graph
==================================================
    val numEdges: Long
    val numVertices: Long
    val inDegrees: VertexRDD[Int]
    val outDegrees: VertexRDD[Int]
    val degrees: VertexRDD[Int]
    // Views of the graph as collections
===============================================
    val vertices: VertexRDD[VD]
    val edges: EdgeRDD[ED]
    val triplets: RDD[EdgeTriplet[VD, ED]]
    // Functions for caching graphs
==================================================
    def  persist(newLevel:  StorageLevel  =  StorageLevel.MEMORY_ONLY):
Graph[VD, ED]
    def cache(): Graph[VD, ED]
    def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
    // Change the partitioning heuristic
==============================================
    def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
    // Transform vertex and edge attributes
============================================
    def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
    def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
    def   mapEdges[ED2](map:   (PartitionID,   Iterator[Edge[ED]])   =>
Iterator[ED2]): Graph[VD, ED2]
    def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
    def   mapTriplets[ED2](map:   (PartitionID,   Iterator[EdgeTriplet[VD,
ED]]) => Iterator[ED2])
```

```
      : Graph[VD, ED2]
    // Modify the graph structure
==================================================
    def reverse: Graph[VD, ED]
    def subgraph(
        epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
        vpred: (VertexID, VD) => Boolean = ((v, d) => true))
      : Graph[VD, ED]
    def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
    def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
    // Join RDDs with the graph
====================================================
    def joinVertices[U](table: RDD[(VertexID, U)])(mapFunc: (VertexID,
VD, U) => VD): Graph[VD, ED]
    def outerJoinVertices[U, VD2](other: RDD[(VertexID, U)])
        (mapFunc: (VertexID, VD, Option[U]) => VD2)
      : Graph[VD2, ED]
    // Aggregate information about adjacent triplets
====================================
    def collectNeighborIds(edgeDirection: EdgeDirection):
VertexRDD[Array[VertexID]]
    def collectNeighbors(edgeDirection: EdgeDirection):
VertexRDD[Array[(VertexID, VD)]]
    def aggregateMessages[Msg: ClassTag](
        sendMsg: EdgeContext[VD, ED, Msg] => Unit,
        mergeMsg: (Msg, Msg) => Msg,
        tripletFields: TripletFields = TripletFields.All)
      : VertexRDD[A]
    // Iterative graph-parallel computation
============================================
    def pregel[A](initialMsg: A, maxIterations: Int, activeDirection:
EdgeDirection)(
        vprog: (VertexID, VD, A) => VD,
        sendMsg: EdgeTriplet[VD, ED] => Iterator[(VertexID,A)],
        mergeMsg: (A, A) => A)
      : Graph[VD, ED]
    // Basic graph algorithms
======================================================
    def pageRank(tol: Double, resetProb: Double = 0.15): Graph[Double,
```

```
Double]
    def connectedComponents(): Graph[VertexID, ED]
    def triangleCount(): Graph[Int, ED]
    def stronglyConnectedComponents(numIter: Int): Graph[VertexID, ED]
  }
```

9.7.3　属性操作

与 RDD 的 map 操作一样，属性图包含下面的操作：

```
  class Graph[VD, ED] {
    def mapVertices[VD2](map: (VertexId, VD) => VD2): Graph[VD2, ED]
    def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
    def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
  }
```

每个操作都产生一个新的图，这个新的图包含 map 操作后的顶点或边的属性。需要提醒读者注意的是，在每次操作下图结构都不受影响。这个重要特征是允许新图形重用原有图形的结构索引（indices）。下面的两行代码在逻辑上是等价的，但是第一个不保存结构索引，所以不会从 GraphX 系统优化中受益。

```
  val newVertices = graph.vertices.map { case (id, attr) => (id,
mapUdf(id, attr)) }
  val newGraph = Graph(newVertices, graph.edges)
```

另一种方法是用 mapVertices 保存索引：

```
  val newGraph = graph.mapVertices((id, attr) => mapUdf(id, attr))
```

9.7.4　结构操作

GraphX 支持一些简单的结构性操作。下面是基本的结构性操作列表。

```
  class Graph[VD, ED] {
    def reverse: Graph[VD, ED]
    def subgraph(epred: EdgeTriplet[VD,ED] => Boolean,
                 vpred: (VertexId, VD) => Boolean): Graph[VD, ED]
    def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
    def groupEdges(merge: (ED, ED) => ED): Graph[VD,ED]
  }
```

reverse 操作返回一个新的图，这个图的边的方向都是反转的。subgraph 操作利用顶点和边的谓词（predicates），返回的图仅仅包含满足顶点谓词的顶点、满足边谓词的边以及满足顶点谓词的连接顶点（connect vertices）。subgraph 操作可以用于很多场景，如获取感兴趣的顶点和边组成的图，或者清除一个图的断链接（broken links）。下面的例子删除了断链接：

```
    // 为顶点创建 RDD
    val users: RDD[(VertexId, (String, String))] =
      sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal",
"postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof")),
(4L, ("peter", "student"))))
    // 为边创建 RDD
    val relationships: RDD[Edge[String]] =
      sc.parallelize(Array(Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"),
Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi"), Edge(4L, 0L, "student"),
Edge(5L, 0L, "colleague")))

    // Define a default user in case there are relationship with missing
user
    val defaultUser = ("John Doe", "Missing")
    //构建初始的 Graph
    val graph = Graph(users, relationships, defaultUser)
    // Notice that there is a user 0 (for which we have no information)
connected to users
    // 4 (peter) and 5 (franklin).
    graph.triplets.map(
        triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of "
+ triplet.dstAttr._1
      ).collect.foreach(println(_))

    // Remove missing vertices as well as the edges to connected to them
    val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 !=
"Missing")
    // The valid subgraph will disconnect users 4 and 5 by removing user
0
    validGraph.vertices.collect.foreach(println(_))
    validGraph.triplets.map(
        triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of "
+ triplet.dstAttr._1
      ).collect.foreach(println(_))
```

mask 操作构造一个子图，这个子图只包含输入参数（图）中包含的顶点和边（类似两个图的交集操作）。这个操作可以和 subgraph 操作相结合，基于另外一个相关图的特征去约束一个图。例如：

```
// 运行关联组件，计算每个顶点的连接组件成员
val ccGraph = graph.connectedComponents() // No longer contains missing field
// 删除丢失的顶点和边
val validGraph = graph.subgraph(vpred = (id, attr) => attr._2 != "Missing")
// Restrict the answer to the valid subgraph
val validCCGraph = ccGraph.mask(validGraph)
```

groupEdges 操作合并多个图中的并行边（即：顶点对之间重复的边）。在大量的应用程序中，并行的边可以合并（它们的权重合并）为一条边从而降低图的大小。

9.7.5 关联（join）操作

在许多情况下，有必要将外部数据加入到图中。例如，我们可能有额外的用户属性需要合并到已有的图中，或者我们可能想从一个图中取出顶点特征加入到另外一个图中。这些任务可以用 join 操作完成。下面列出的是主要的 join 操作。

```
class Graph[VD, ED] {
  def joinVertices[U](table: RDD[(VertexId, U)])(map: (VertexId, VD, U) => VD)
    : Graph[VD, ED]
  def outerJoinVertices[U, VD2](table: RDD[(VertexId, U)])(map: (VertexId, VD, Option[U]) => VD2)
    : Graph[VD2, ED]
}
```

joinVertices 操作将输入 RDD 和顶点相结合，返回一个新图。新图的顶点属性是使用用户定义的 map 函数获得的。在 RDD 中没有匹配值的顶点保留其原始值。需要注意的是，对于给定的顶点，如果 RDD 中有超过 1 个的匹配值，则仅仅使用其中的一个。建议使用下面的方法保证输入 RDD 的唯一性。下面的方法也会预索引返回的值，用以加快后续的 join 操作：

```
val nonUniqueCosts: RDD[(VertexID, Double)]
val uniqueCosts: VertexRDD[Double] =
  graph.vertices.aggregateUsingIndex(nonUnique, (a,b) => a + b)
val joinedGraph = graph.joinVertices(uniqueCosts)(
```

```
    (id, oldCost, extraCost) => oldCost + extraCost)
```

outerJoinVertices 与 joinVertices 类似，区别是前者将用户自定义的 map 函数用到所有顶点和改变顶点属性类型。这是因为并不是所有顶点在 RDD 中拥有匹配的值。map 函数需要一个 option 类型。比如：

```
  val outDegrees: VertexRDD[Int] = graph.outDegrees
  val degreeGraph = graph.outerJoinVertices(outDegrees) { (id, oldAttr, outDegOpt) =>
    outDegOpt match {
      case Some(outDeg) => outDeg
      case None => 0 // No outDegree means zero outDegree
    }
  }
```

9.7.6 聚合操作

图分析任务的一个关键步骤是汇总每个顶点的临接点的信息。例如：我们可能想知道每个用户的追随者的数量或者每个用户的追随者的平均年龄。许多迭代图算法（如 PageRank、最短路径、连通体）多次聚合相邻顶点的属性。为了提高性能，主要的聚合操作从 graph.mapReduceTriplets 改为了新的 graph.AggregateMessages。

GraphX 中的核心聚合操作是 aggregateMessages。这个操作将用户定义的 sendMsg 函数应用到图的每个边三元组（edge triplet），然后应用 mergeMsg 函数在其目的顶点聚合这些消息。其定义如下：

```
  class Graph[VD, ED] {
    def aggregateMessages[Msg: ClassTag](
        sendMsg: EdgeContext[VD, ED, Msg] => Unit,
        mergeMsg: (Msg, Msg) => Msg,
        tripletFields: TripletFields = TripletFields.All)
      : VertexRDD[Msg]
  }
```

用户自定义的 sendMsg 发送消息给源和目的属性。可将 sendMsg 函数看做 map-reduce 过程中的 map 函数。mergeMsg 函数把到相同的顶点的两个消息合并为一个消息。可以将 mergeMsg 函数看做 map-reduce 过程中的 reduce 函数。 aggregateMessages 操作返回一个包含每个顶点的聚合消息的 VertexRDD[Msg]。没有接收到消息的顶点不包含在返回的 VertexRDD 中。另外，aggregateMessages 有一个可选的 tripletFields 参数，它指定：在 EdgeContext 中，哪些数据被访问（如源顶点属性而不是目的顶点属性）。tripletsFields 可能的选项被定义在 TripletFields 中。tripletFields 参数可被用来指定 GraphX 只需要使用 EdgeContext 上的一部分。例如，如果我们想

计算每个用户的追随者的平均年龄，我们仅仅只需要源字段。所以我们使用 TripletFields.Src 表示我们仅仅只需要源字段。在下面的例子中，我们用 aggregateMessages 操作计算每个用户的资深追随者的平均年龄。

```
    // 导入随机图生成库
    import org.apache.spark.graphx.util.GraphGenerators
    // 创建一个图，顶点属性为 age
    val graph: Graph[Double, Int] =
      GraphGenerators.logNormalGraph(sc,             numVertices             =
100).mapVertices( (id, _) => id.toDouble )
    // 计算 older followers 的人数和总年龄
    val olderFollowers: VertexRDD[(Int, Double)] =
graph.aggregateMessages[(Int, Double)](
      triplet => { // Map 函数
        if (triplet.srcAttr > triplet.dstAttr) {
          // 发送信息到目标 vertex
          triplet.sendToDst(1, triplet.srcAttr)
        }
      },
      // 汇总计数和年龄
      (a, b) => (a._1 + b._1, a._2 + b._2) // Reduce 函数
    )
    // 计算平均年龄
    val avgAgeOfOlderFollowers: VertexRDD[Double] =
      olderFollowers.mapValues( (id, value) => value match { case (count,
totalAge) => totalAge / count } )
    // 显示结果
    avgAgeOfOlderFollowers.collect.foreach(println(_))
```

9.7.7　计算度信息

最一般的聚合任务就是计算顶点的度，即每个顶点相邻边的数量。在有向图中，经常需要知道顶点的入度、出度以及总共的度。GraphOps 类包含一个操作集合，用来计算每个顶点的度。例如，下面的例子计算最大的入度、出度和总度。

```
    // Define a reduce operation to compute the highest degree vertex
    def max(a: (VertexId, Int), b: (VertexId, Int)): (VertexId, Int) = {
      if (a._2 > b._2) a else b
    }
```

```
// Compute the max degrees
val maxInDegree: (VertexId, Int)  = graph.inDegrees.reduce(max)
val maxOutDegree: (VertexId, Int) = graph.outDegrees.reduce(max)
val maxDegrees: (VertexId, Int)   = graph.degrees.reduce(max)
```

9.7.8 缓存操作

在 Spark 中，RDD 默认是不缓存的。为了避免重复计算，当需要多次利用它们时，我们必须显式地缓存它们。GraphX 中的图也有相同的方式。当利用到图多次时，确保首先执行 Graph.cache()方法。

在迭代计算中，为了获得最佳的性能，可能需要不缓存。默认情况下，缓存的 RDD 和图会一直保留在内存中，直到因为内存压力迫使它们以 LRU 的顺序删除。对于迭代计算，前一次迭代的中间结果填充到缓存中。虽然它们最终会被删除，但是在内存中缓存不需要的数据将会减慢垃圾回收。如果中间结果不需要，不缓存它们是更加高效的。对于迭代计算，我们建议使用 Pregel API，它可以正确地选择不持久化中间结果。

9.7.9 图算法

GraphX 提供了一套图算法工具包，方便用户对图进行分析。目前最新版本已支持 PageRank 、三角形计数、最大连通图和最短路径等图算法。下面我们以 PageRank（即网页排名）为例讲解图算法。

PageRank 又称网页级别，它是 Google 创始人拉里 • 佩奇和谢尔盖 • 布林于 1997 年构建早期的搜索系统原型时提出的链接分析算法。目前很多重要的链接分析算法都是在 PageRank 算法基础上衍生出来的。PageRank 是 Google 用来标识网页的相关性或重要性的一种方法，是 Google 用来衡量一个网站的好坏的唯一标准。在揉合了诸如 Title 标识和 Keywords 标识等所有其他因素之后，Google 通过 PageRank 来调整结果，使那些更具“等级 / 重要性”的网页在搜索结果中排在前面，从而提高搜索结果的相关性和质量。PageRank 通过网络链接关系来确定一个页面的等级。Google 把从 A 页面到 B 页面的链接解释为 A 页面给 B 页面投票，Google 根据投票来源（甚至是来源的来源）和投标目标的等级来决定新的等级。简单来说，一个高等级的页面可以使低等级页面的等级提升。

在社交网络数据集中，我们可以使用这个算法计算每个用户的网页级别。在下面的例子中，我们有一组用户数据，一组用户之间的关系数据，计算过程如下：

```
// 装载边信息
val graph = GraphLoader.edgeListFile(sc, "graphx/data/followers.txt")
// 运行 PageRank
val ranks = graph.pageRank(0.0001).vertices
// 关联用户名
```

```
val users = sc.textFile("graphx/data/users.txt").map { line =>
  val fields = line.split(",")
  (fields(0).toLong, fields(1))
}
val ranksByUsername = users.join(ranks).map {
  case (id, (username, rank)) => (username, rank)
}
// 打印结果
println(ranksByUsername.collect().mkString("\n"))
```

第 10 章 ◀ 大数据分析 ▶

在现实世界中，一些公司正在用数千个特征和数十亿个交易来构建信用卡欺诈检测模型，另一些公司正在向数百万用户智能地推荐数百万产品。这些都是大数据分析的范畴。大数据分析可以用来解决海量数据、异构数据等多种问题带来的数据分析难题。

大数据处理包含以下几个方面：数据采集、数据转换和清洗、数据存储、数据检索、数据分析等。根据数据处理的不同阶段，有不同的专业工具来对数据进行不同阶段的处理。在数据转换部分，有专业的 ETL 工具来帮助完成数据的清洗、提取、转换和加载。去掉无关的数据和不重要的数据，对数据进行相关分类。在数据存储和计算部分的技术，有 Hadoop 等。在数据可视化部分，需要对数据的计算结果进行分析和展现。一般而言，在分类划分之后，就可以根据具体的分析需求选择模式分析的技术，如路径分析、兴趣关联规则、聚类等。通过模式分析，找到有用的信息。

大数据分析包括：（1）数据挖掘算法：大数据分析的理论核心就是数据挖掘算法，各种数据挖掘的算法基于不同的数据类型和格式才能更加科学地呈现出数据本身的特点，也正是因为这些被全世界统计学家所公认的各种统计方法才能深入数据内部，挖掘出公认的价值。另外一个方面也是因为有这些数据挖掘的算法才能更快速地处理大数据，如果一个算法得花上好几年才能得出结论，那大数据的价值也就无从说起了。（2）大数据预测性分析：大数据分析最终要应用的领域之一就是预测性分析，从大数据中挖掘出特点，通过科学的建立模型，之后便可以通过模型带入新的数据，从而预测未来的数据。（3）可视化分析：大数据分析的使用者有大数据分析专家，同时还有普通用户，但是他们二者对于大数据分析最基本的要求都是能够可视化分析，因为可视化分析能够直观地呈现大数据特点，同时能够非常容易被读者所接受，就如同看图说话一样简单明了。

还有一点，大数据分析系统的体系架构正发生着根本性的变革。过去 10 年，大多由 IT 部门主导大数据分析项目，这些项目高度可控、中心化、IT 化。现在，大量的商业用户迫切要求进行交互式分析，希望通过深度分析获取数据洞察力，而他们只有非常有限的 IT 或数据科学技能。一方面 IT 部门需要满足越来越多的 Data Discovery 需求，另一方面他们又不想牺牲可控性。无论是 IT 部门主导的数据分析，还是业务部门主导的，大数据分析分为以下几步：

1. 确定分析目标

这些分析目标可能是由一些问题组成。比如：一个企业的产品销路不好。那么，你可能问：

“价格太高了？”“竞争对手的产品有无独到之处？”“竞争对手的产品的目标客户是谁？”等等。大数据采集依赖于这些问题。比如，你可能需要采集客户的反馈信息，竞争对手的产品规格，等等。总之，我们首先要确定一个清晰的、可评估的目标。

2. 设置评估维度

在数据采集之前一定要确定评估参数。常见的评估参数有：时间维度，评估因子等。

3. 采集数据

原始数据来自不同的数据源。有一个容易让技术人员忽视的因素是数据采集的预算。比如，银行可能需要运营商的消费数据，从而从一个维度来评估信用卡申请人员的消费金额和信用信息。这样的数据往往需要外部采购来获得。

4. 数据清洗

垃圾数据必然影响分析的质量，所以，我们需要清洗数据。一般情况下，这需要一个自动化的业务流程来清洗。

5. 数据分析

有不同的数据分析技术，如：探索性数据分析技术、描述统计法、数据可视化等。

10.1 数据科学

在大数据领域，我们可以把众多的大数据工具（如：Spark、Hadoop 等）比作厨房工具，而数据本身可以比作原材料。那么，只有工具和好的原材料未必能做出来一个好菜，这其中缺少一个优秀的厨师。数据科学家就是大数据领域的厨师。数据科学就是利用大数据工具将原始数据变成对不懂数据科学的普通人有价值的东西。数据科学界有几个常识，供大家参考：

（1）成功的大数据分析中绝大部分工作是数据预处理（数据整合）。数据是在不同的系统上，本身还可能是混乱的，在让数据产生价值之前，必须对数据进行清洗、转换、融合、存储和管理。我们有时需要花大量的时间在特征提取和选择上。比如，在信用卡欺诈上，数据科学家需要从许多可能的特征中进行选择。在将这些特征转换成适用于机器学习算法的向量时，每个特征可能会有不同的问题。

（2）迭代与数据科学紧密相关。建模和分析经常要对一个数据集进行多次遍历。数据科学家本身的工作流程也涉及迭代，比如：数据科学家往往很难在第一次就得到理想的结果。选择正确的特征，挑选合适的算法，运行恰当的测试，所有这些工作都需要反复试验。

（3）构建好的模型只是成功的一步，而不是全部。模型往往需要定期重建。

在大数据分析场景中，我们要区分试验环境和生产环境。对于试验环境，数据科学家进行探

索性分析，他们用各种特征做试验，用辅助数据源来增强数据，他们试验各种算法，希望从中找到一两个有效算法。而对于生产环境，数据科学家进行操作式分析。他们把模型打包成服务，这些服务可以作为决策依据。他们跟踪模型随时间的表现，精心调整模型。值得指出的是，Spark 技术在探索型分析系统和操作型分析系统之间搭起一座桥梁。

10.1.1 探索性数据分析

探索性数据分析（Exploratory Data Analysis，简称 EDA）是指对已有的原始数据在尽量少的先验假定下进行探索，通过作图、制表、方程拟合、计算特征量等手段探索数据的结构和规律的一种数据分析方法。特别是当我们对这些数据中的信息没有足够的经验，不知道该用何种传统统计方法进行分析时，探索性数据分析就会非常有效。探索性数据分析在 20 世纪六十年代被提出，其方法由美国著名统计学家约翰 • 图基（John Tukey）命名。

EDA 的出现主要是在对数据进行初步分析时，往往还无法进行常规的统计分析。这时候，如果分析者先对数据进行探索性分析，辨析数据的模式与特点，并把它们有序地发掘出来，就能够灵活地选择和调整合适的分析模型，并揭示数据相对于常见模型的种种偏离。在此基础上再采用以显著性检验和置信区间估计为主的统计分析技术，就可以科学地评估所观察到的模式或效应的具体情况。概括起来说，分析数据可以分为探索和验证两个阶段。探索阶段强调灵活探求线索和证据，发现数据中隐藏的有价值的信息，而验证阶段则着重评估这些证据，相对精确地研究一些具体情况。在验证阶段，常用的方法是传统的统计学方法，在探索阶段，主要的方法就是 EDA，下面我们重点对 EDA 做进一步的说明。

EDA 的特点有三个：

- 在分析思路上让数据说话。传统统计方法通常是先假定一个模型，例如数据服从某个分布（特别常见的是正态分布），然后使用适合此模型的方法进行拟合、分析及预测。但实际上，多数数据并不能保证满足假定的理论分布。因此，传统方法的统计结果常常并不令人满意，使用上受到很大的局限。EDA 则可以从原始数据出发，深入探索数据的内在规律，而不是从某种假定出发，套用理论结论，拘泥于模型的假设。
- EDA 分析方法灵活，而不是拘泥于传统的统计方法。传统的统计方法以概率论为基础，使用有严格理论依据的假设检验、置信区间等处理工具。EDA 处理数据的方式则灵活多样，分析方法的选择完全从数据出发，灵活对待，灵活处理，什么方法可以达到探索和发现的目的就使用什么方法。这里特别强调的是 EDA 更看重的是方法的稳健性、耐抗性，而不刻意追求概率意义上的精确性。
- EDA 分析工具简单直观，更易于普及。传统的统计方法都比较抽象和深奥，一般人难于掌握，EDA 则更强调直观及数据可视化，更强调方法的多样性及灵活性，使分析者能一目了然地看出数据中隐含的有价值的信息，显示出其遵循的普遍规律以及与众不同的突出特点，促进发现规律，得到启迪，满足分析者的多方面要求，这也是 EDA 对于数据分析的主要贡献。

10.1.2　描述统计

描述统计用来描绘（describe）或总结（summarize）所采集到的数据，通过图表形式对所收集的数据进行加工处理和显示，进而通过综合概括与分析得出反映客观现象的规律性数量特征。常用的工具有：平均数（Mean）、中位数（Median）、众数（Mode）、几何平均数、全距（range）、平均差（average deviation）、标准差（standard deviation）、相对差、四分差（quartiledeviation）等。

10.1.3　数据可视化

有时，我们辛辛苦苦分析一堆大数据，得到很多报表，但是客户没看懂！如果你正着手于从数据中洞察出有用信息，那你需要“数据可视化”。俗话说，一图胜千言。良好的可视化帮助用户获取数据的多维度透视视图。

数据可视化是指将大型数据集中的数据以图形、图像形式表示，并利用数据分析和开发工具发现其中未知信息的处理过程。数据可视化工具基本以表格、图形（chart）等可视化元素为主，数据可进行过滤、钻取、数据联动、跳转、高亮等分析手段做动态分析。可视化工具可以提供多样的数据展现形式、多样的图形渲染形式、丰富的人机交互方式，支持商业逻辑的动态脚本引擎等等。

基于 Reporting 去指导业务的需求虽然还存在，目前最显著的改变却是如何二者兼之，尤其是满足新的 Business-user-driven（业务用户驱动）的需求。这些需求不再使用传统的、IT-centric 的企业级平台，转而采用去中心化的 Data Discovery 部署，如今这种部署在企业里随处可见。Gartner 估算，超过 1/2 的购买需求来自于 Data-discovery-driven（数据发现驱动）。这种去中心化模型让更多商业用户获取到了数据分析能力，同时也产生了对可控的 Data Discovery 方法的需求。这是一个持续了多年的转变。IT-centric BI 平台正越来越多地被 Business-user-driven 和交互式分析项目替换。这个转变的目标，是让更大范围的用户和更多的场景能获取到数据分析能力。

随着企业通过可管控的 Data Discovery 方法建设 BI 平台，很多商业用户以 Self-service（自服务）的模式去访问 IT 部门把控的数据源。当前的趋势是，更大范围地接入用户尤其是非传统 BI 用户，以扩展数据分析的应用尤其是通过深度分析产生洞察。很多数据分析整合来自内部和外部的多结构化数据。对 BI 厂商来说，整合线上线下的、多结构化的、流式的数据，已经成为很重要的功能。还要支持社交和网络分析、情绪分析、机器学习。新的挑战和机会来自于将这些多源数据融合并管理起来，以产生商业价值。

本节将聚焦在一些通用的数据库可视化技术，帮助您打造可视化层。下面是一些基本原则：

- 确保可视化层显示的数据都是从最后的汇总输出表中取得的数据。这么做可以避免直接读取整个原始数据。这不仅最大限度地减少数据传输，而且当用户在线查看报告时还有助于避免性能卡顿问题。
- 充分利用可视化工具的缓存。缓存可以对可视化层的整体性能有提升。

- 物化视图是可以提高性能的另一个重要的技术。
- 大部分可视化工具允许通过增加线程数来提高请求响应的速度。如果资源足够、访问量较大时，这是提高系统性能的好办法。
- 尽量提前将数据进行预处理，如果一些数据必须在运行时计算，则将运行时计算简化到最小。
- 可视化工具可以按照各种各样的展示方法对应不同的读取策略，这包括：离线模式或者在线连接模式。每种服务模式都是针对不同场景设计的。

近年来，随着大数据时代的来临，数据可视化产品已经不再满足于使用传统的数据可视化工具来对数据仓库中的数据抽取、归纳并简单地展现。大数据可视化产品必须快速地收集、筛选、分析、归纳、展现决策者所需要的信息，并根据新增的数据进行实时更新。因此，在大数据时代，数据可视化工具必须具有以下特性：

- 实时性：数据可视化工具必须适应大数据时代数据量的爆炸式增长需求，必须快速地收集分析数据，并对数据信息进行实时更新；
- 简单操作：数据可视化工具满足快速开发、易于操作的特性，能满足互联网时代信息多变的特点；
- 更丰富的展现：数据可视化工具需要具有更丰富的展现方式，能充分满足数据展现的多维度要求。

企业获取数据可视化功能主要通过编程和非编程两类工具实现。主流编程工具包括以下三种类型：从艺术的角度创作的数据可视化，比较典型的工具是 Processing.js，它是为艺术家提供的编程语言。从统计和数据处理的角度，R 语言是一款典型的工具，它本身既可以做数据分析，又可以做图形处理。介于两者之间的工具，既要兼顾数据处理，又要兼顾展现效果，D3.js 是一个不错的选择。像 D3.js 这种基于 JavaScript 的数据可视化工具更适合在互联网上互动地展示数据。D3.js 是数据驱动文件（Data-Driven Documents）的缩写，它通过使用 HTML/CSS 和 SVG 来渲染精彩的图表和分析图。D3 对网页标准的强调足以满足使其在所有主流浏览器上都能使用的可能性，它可以将视觉效果很棒的组件和数据驱动方法结合在一起。

非编程类工具除了微软公司的 Excel 之外，还有：

- FusionCharts：不仅有漂亮的图表，还能制作出生动的动画、巧妙的设计和丰富的交互性。它在 PC 端、Mac、iPad、iPhone 和 Android 平台都可兼容，具有很好的用户体验一致性，同时也适用于所有的网页和移动应用。FusionCharts 套件提供了超过 90 种图表和图示，例如：漏斗图、热点地图、放缩线图和多轴图等。
- Dygraphs：这是一款快捷灵活的开源 JavaScript 图表库。它具有极强的交互性，比如缩放、平移和鼠标悬停等都是默认动作。Dygraphs 也是高度兼容的，所有的主流浏览器都可正常运行。你还可以在手机和平板设备上使用双指缩放！
- Datawrapper：让你只需 4 步就可以创建出图表和地图。这款工具帮你将数据可视化的时间从几小时减少到了几分钟。它的操作非常简单，你只需上传数据，选择一个图表

或地图，然后点击发布就可以了。Datawrapper 是为你的需求定制化而存在的，版式和视觉效果都可以按照你的样式规范而调整。

- Leaflet：这是为移动端交互地图所做的开源 JavaScript 库，其中包含了大部分在线地图开发人员都需要的特征。Leaflet 被设计为一个简单易用、性能优良的工具。归功于 HTML5 和 CSS3，它得以支持所有主流电脑和移动平台。它还有大量可供选择的插件能安装。
- Tableau Public：是一款操作简便的 app，它可以轻松帮你创建可视化作品。只需打开数据，用 Tableau 桌面版来进一步探索。然后，把可视化内容存储在你 Tableau Public 在线文件空间，最后，将他们放进网站或 Blog 即可。
- Google Charts：能为你的网站提供完美的数据可视化处理。从简单的折线图到复杂的分级树形图，它的图表库里提供了海量的模版可供选择。所有的图表样式都是使用数据库表类（DataTable class）来填充数据的，这意味着你可以在挑选表现效果的时候轻松转换表格类型。
- Raw：是一款开放的 Web app，可以按需创作矢量图形可视化作品。它是使用 LGPL 许可的定制项目，允许随意下载并修改。但是 Raw 只是 Web app，你所上传的数据只能用网页浏览器处理，因此没有实质性服务器端数据交互。你可将可视化作品导出为矢量图形（SVG）格式或者 PNG 格式。
- iCharts：是基于云端的趋势预测视觉分析平台，它可以快速地将复杂的商业信息、大规模调查数据和动态数据研究的结果可视化。它非常得快速简便，但却可以根据实时数据创造出富有冲击力的视觉智能图像，并且可以为你带来全方位信息聚合和信息对比。
- HighCharts：通过 HighCharts 你可以为网站项目制作交互式图表。它的用户非常广泛（全世界最大的 100 家公司里面有 61 家以及成千上万的开发人员都在使用）。HighCharts 是建立在 HTML 5 上的，在现代的浏览器包括移动、平板设备上运行，也支持过时的 IE 浏览器（IE 6 之后的都可以）。它同时也是动态的，你可以自由添加、移除、修改数据列（Series）和关键点（Points）。这款 app 支持多种类型的图表：折现图、样条曲线、面积图、曲线面积图、柱状图、条状图、饼状图和散点图等等。
- InstantAtlas：InstantAtlas 可以创建交互式动态分配图报告，并结合统计数据和地图数据来优化数据可视化效果。
- Visual.ly：是一个综合图库和信息图表生成器。它的工具很简单，却可创造出不错的数据展示作品。
- Polymaps：是一个可以同时使用位图和 SVG 矢量地图的 JavaScript 库。它为地图提供的多级缩放数据集，并且可支持矢量数据的多种视觉表现形式。

10.2 预测分析

预测分析就是挖掘采集来的海量数据，从中预测未来的行为模式和趋势。金融行业给每个客户的信用打分（credit score）就是预测分析的一个使用场景。除了金融行业，其他行业的预测产品也非常火热。比如：2015 年 IBM 购买了一家专门从事预测分析的医疗数据公司 Explorys。

预测分析的一个最经典的例子就是英国科学家高尔顿在 19 世纪发现的趋均数回归（regression to the mean）。他按大小将豌豆种子分开种植，他发现，虽然下一代的种子往往和父辈很相像，但总体来看，他们的平均大小更接近平均水平。一个异常的结果后将会紧跟着出现一个预期接近平均值的结果，这被称为“均值回归”。如果这种均值回归不存在的话，那么大的豌豆就会繁殖出更大的豌豆，小的豌豆就会繁殖出更小的豌豆，如此这样，这个世界就会只有侏儒和巨人。大自然会使每一代变得愈发畸形，最终达到我们无法想象的极端。

趋均数回归就是早期的预测模型，它意味着知道了一个值就大体知道了另一个值。如果知道一颗新豌豆的大小，根据这种关联关系，我们就能更准确的估计其后代的大小。经过一百多年统计学的发展，随着现代机器学习的出现，我们依旧把以“某些值”预测“另外某个值”的思想称为回归，即使它已经和“向均数回归”没有任何关系。“回归”是预测一个数值型数量，比如：大小、收入等。而“分类”是预测标签（label），即类别。比如判断某个邮件为垃圾邮件。回归和分类都是通过一个或更多值预测另一个（或更多）值。为了能够做出预测，两者都需要从一组输入和输出中学习预测规则。在学习过程中，需要告诉它们问题以及问题的答案，它们都属于监督学习的范畴。

10.2.1 预测分析实例

预测分析是运用各种定性和定量的分析理论与方法，对事物未来发展的趋势和水平进行判断和推测的一种活动。预测分析包含了多种模型，其中最常用的模型是预测模型。我们来看一个实际的例子。随着互联网的发展，人们越来越习惯于在网上搜索电影信息。谷歌发现，电影相关的搜索量与票房收入之间存在很强的关联。据此，谷歌公布了电影票房预测模型，这个预测模型是大数据分析技术在电影行业的一个重要应用。该模型能够提前一个月预测电影上映首周的票房收入，准确度高达 94%。谷歌票房预测模型的基础是将电影相关的搜索量与票房收入的关联。谷歌采用了如下三类指标：

- 电影预告片的搜索量
- 同系列电影前几部的票房表现
- 档期的季节性特征

其中每类指标又包含了多项类内指标。在获取到每部电影的这些指标后，谷歌构建了一个线性回归模型（linear regression model）来建立这些指标和票房收入的关系。图 10-1 展示了模型的

效果，横轴是预告片搜索量，纵轴是首周票房收入，灰色圆点代表了实际的票房收入，红色方块代表了预测的票房收入。可以看到，预测结果与实际结果非常接近。

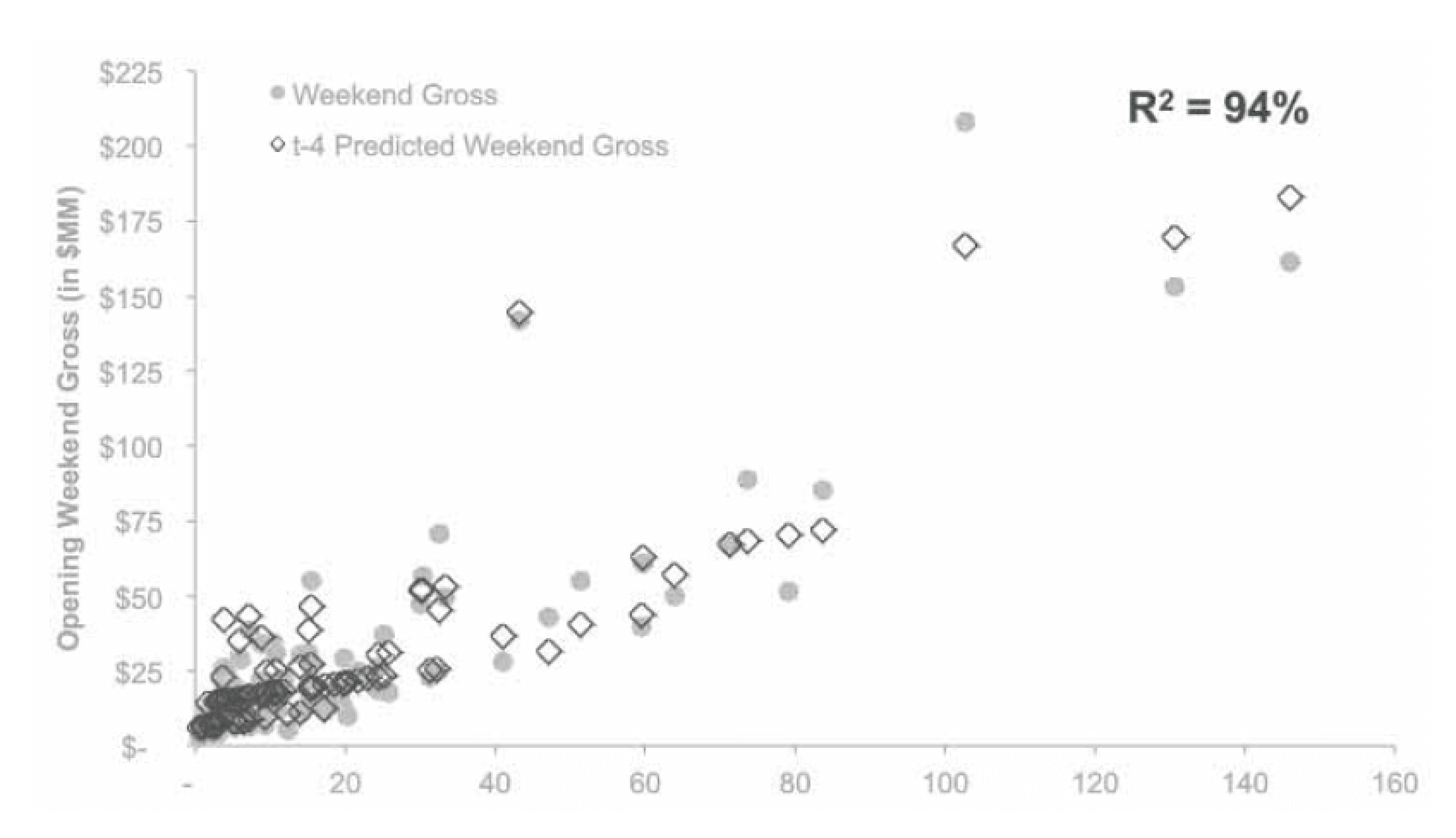


图 10-1　提前一个月预测票房的效果

谷歌采用的是数据分析中最简单的模型之一：线性回归模型。首先，线性模型虽然简单，但已经达到了很高的准确度（94%）。简单且效果好，是我们在实际应用中一直追求的。其次，简单的模型易于被人们理解和分析。大数据分析技术的优势正是能够从大量数据中挖掘出人们可以理解的规律，从而加深对行业的理解。正是因为谷歌使用了线性预测模型，所以它很容易对各项指标的影响做出分析。例如谷歌的报告中给出了这样的分析结论："距离电影上映一周的时候，如果一部影片比同类影片多获得 25 万搜索量，那么该片的首周票房就很可能比同类影片高出 430 万美元"。对于电影的营销来说，掌握各项指标对票房收入的影响，可以优化营销策略，降低营销成本。谷歌的报告中指出，用户一般会通过多达 13 个渠道来了解电影的信息。票房预测模型的出现无疑使得营销策略的制定更加有效。票房预测模型的公布，让业内人士再次见证了大数据的成功应用。近年来，大数据在电影行业的应用越来越引起关注，比如此前谷歌利用搜索数据预测了奥斯卡获奖者。

大数据分析的本质，在于通过数据，更精准地挖掘用户的需求。而谁能掌握用户的需求，谁就可以引领行业的发展。谷歌的票房预测模型，本质上也是通过搜索量，挖掘出用户对电影的需求有多大，进而预测出票房收入。值得注意的是，谷歌的模型基于的只是宏观的搜索量的统计，对用户需求的挖掘相对表面。如何从搜索数据中更深地挖掘用户的需求将是未来的趋势之一。

既然大数据分析的核心是挖掘用户需求，所以一大核心问题是：哪些用户的需求是可以从数据中挖掘到的？要知道，并不是任何需求都可以被挖掘到，或者说可以被精准地挖掘到。能够通过大数据分析挖掘到的需求，一般是符合行业经验的，应当是业内人士觉得可以被挖掘的（有时候，挖掘出的需求可能会超出行业经验，甚至产生颠覆性的影响）。谷歌的预测模型的基本假设，是符合行业直觉的，即电影的搜索量越大，往往票房收入越大。模型能够提前一个月预测票房，也符合行业经验，正如谷歌的一项行业调研揭示的：大多数观众会在电影首映 4 周前去了解电影。数据分析技术，是把这种模糊的行业经验，变得更科学，变得更精准。而这一过程，很可

能会深层次地改变电影行业。

10.2.2 回归（Regression）分析预测法

回归分析预测法是在分析某一个现象的自变量和因变量之间相关关系的基础上，建立变量之间的回归方程，并将回归方程作为预测模型，根据自变量在预测期的数量变化来预测因变量，因此，回归分析预测法是一种重要的预测方法。当我们在对某一个现象未来发展状况和水平进行预测时，如果能将影响预测对象的主要因素找到，并且能够取得其数量资料，就可以采用回归分析预测法进行预测。它是一种具体的、行之有效的、实用价值很高的常用市场预测方法。

回归分析预测法有多种类型。依据相关关系中自变量的个数不同分类，可分为一元回归分析预测法和多元回归分析预测法。在一元回归分析预测法中，自变量只有一个，而在多元回归分析预测法中，自变量有两个以上。依据自变量和因变量之间的相关关系不同，可分为线性回归预测和非线性回归预测。回归分析预测法的步骤是：

1. 根据预测目标，确定自变量和因变量

明确预测的具体目标，也就确定了因变量。如预测具体目标是下一年度的销售量，那么销售量 Y 就是因变量。通过市场调查和查阅资料，寻找与预测目标的相关影响因素，即自变量，并从中选出主要的影响因素。

2. 建立回归预测模型

依据自变量和因变量的历史统计资料进行计算，在此基础上建立回归分析方程，即回归分析预测模型。

3. 进行相关分析

回归分析是对具有因果关系的影响因素（自变量）和预测对象（因变量）所进行的数理统计分析处理。只有当变量与因变量确实存在某种关系时，建立的回归方程才有意义。因此，作为自变量的因素与作为因变量的预测对象是否有关，相关程度如何，以及判断这种相关程度的把握性多大，就成为进行回归分析必须要解决的问题。进行相关分析，一般需要求出相关关系，以相关系数的大小来判断自变量和因变量的相关程度。

4. 检验回归预测模型，计算预测误差

回归预测模型是否可用于实际预测，取决于对回归预测模型的检验和对预测误差的计算。回归方程只有通过各种检验，且预测误差较小，才能将回归方程作为预测模型进行预测。

5. 计算并确定预测值

利用回归预测模型计算预测值，并对预测值进行综合分析，确定最后的预测值。

回归分析预测法是一类比较经典，也比较实用的预测方法。正是由于它比较经典，因此也比较成熟，再加上比较容易理解，运用也就比较广泛。相比之下，其中的线性回归预测法和非线性

回归预测法的运用更广些。在实际使用过程中，如果在选择具体的方法和模型时，能对数据作较为详细的分析，预测结果也就会比较令人满意的。

10.3 机器学习

2016 年 3 月的人机围棋大战突显了机器学习和机器思考的威力。其实，如今的商业决策越来越依赖于预测数据。与数据预测相关的数据科学分支中有不少大家所熟知的方法：机器学习、预测分析和人工智能。在业界，机器学习和预测分析这两个术语有时被交替使用。对于机器学习而言，有很多与其相关的用例：

- 产品推荐
- 客户流失预测
- 欺诈监测和预防
- 信用和风险管理

在预测分析中，我们要对数据执行操作和深入分析两个步骤。前面的几个章节中，我们主要关注的是数据操作，Storm、Spark、Hive 都是不错的数据操作工具。本节讲述数据深入分析的内容。从高层次来看，有两类机器学习算法，分别是监督学习和无监督学习。这两种算法的区别是训练模型和构建模型的方法的不同。对于监督学习来说，其训练集包括特征和目标。对于无监督学习来说，其构建模型不需要给定目标值，而是使用算法在数据中寻找目标。机器学习的基本假设是：过去发生的事情在将来也会以类似的方式发生，许多机器学习算法尝试提取那些数据特征集中隐藏的概念，然后使用这些概念来预测未来相似的事件。

在维基百科上对机器学习提出以下几种定义：

- 机器学习是一门人工智能的科学，该领域的主要研究对象是人工智能，特别是如何在经验学习中改善具体算法的性能。
- 机器学习是对能通过经验自动改进的计算机算法的研究。
- 机器学习是用数据或以往的经验，以此优化计算机程序的性能标准。

为了帮助大家理解什么是机器学习，让我们先来看一个简单的故事。假定我们有一个诊疗机器人，它负责看病，下面可能是机器和病人的一段“对话”：

机器人：您哪不舒服？
患者：发热
机器人弹出两百多个诊断。

患者：最高39度多。
机器人删掉十几个诊断，还剩一百九十多个。

机器人：早上发热还是下午还是晚上？
患者：傍晚时候最明显，早上不怎么发热。
机器人又删掉十几个诊断，还剩一百七十多。

机器人：咳嗽吗？
患者：不咳。
机器人又删掉十几个诊断，还剩一百六十多。
…………

症状描述完后，机器人共列举了48个诊断。

机器人：好的，现在请您进行血常规、尿常规、大便常规、肝功能、肾功能、血糖、血电解质、凝血功能、超敏 C 反应蛋白、降钙素原、免疫全套、抗核抗体、抗中性粒细胞胞浆抗体、血沉、感染四项、血培养、痰培养、尿培养、头部 CT、胸部 CT、上腹部 CT……检查，鉴别诊断。
患者：啊!!!

显然，上面的诊疗机器人会给患者带来很多麻烦。这个机器人还需要更多地模拟医生的学习行为，获取新的知识，不断改善自身的性能。最著名的诊疗机器人当属 IBM 于 2007 年开始研发的沃森（Watson）机器人，沃森运用大量的临床病例，可在短时间内分析可能的结果，并协助医生做出治疗建议，大大减少医生疏忽的机会。沃森不仅可以即时让医师参考诊断与治疗方式，针对可能的疾病做深入的问诊，还可以有效减少医疗纠纷，缩小判断误差。

机器学习包含算法、模型和评估三个部分。机器学习是数据通过算法构建出模型并对模型进行评估，评估的性能如果达到要求就拿这个模型来测试其他的数据，如果达不到要求就要调整算法来重新建立模型，再次进行评估，如此循环往复，最终获得满意的经验来处理其他的数据。

10.3.1 机器学习的市场动态

大数据分析市场的一个问题是，有些企业尤其是中小企业还未建立起对数据的正确认识，不太了解数据的真正价值，也不知道如何通过数据来指导运营和业务，这需要一个中长期的培育。而另一方面，我们也看到除了大型机构和大型企业之外，一部分中小企业非常清晰地认识到数据分析的价值，具有非常强烈地建立有效的数据化运营体系的愿望，他们广泛地分布在电商、金融、O2O 等泛互联网行业，他们预测用户行为，并推荐相关产品，提供危险交易预警服务等。我们已经注意到，越来越多的企业构建适配的数据分析平台，充分发掘数据价值，快速成长为所处行业的佼佼者。下面我们分析 IBM 公司在机器学习上的布局，以此了解整个大数据分析的市场趋势。

在 2015 年，IBM 收购了初创公司 AlchemyAPI，旨在利用其工具加强 Watson 人工智能。AlchemyAPI 提供深度学习服务等技术。IBM 还设立了 Watson Health，并收购了几家规模较小的

医疗数据公司，跟苹果、强生和 Medtronic 建立合作关系以收集可穿戴设备的数据。同时，IBM 与美敦力合作改善糖尿病疾病管理。通过运用美敦力的设备、护理管理产品、疗法和辅导以及 IBM 的沃森医疗云平台以优化患者的治疗效果。之后，IBM 收购了两家初创公司：Explorys（一家可以查看 5000 万份美国患者病例的分析公司）和 Phytel（提供软件把各种类型的健康数据进行处理，为医生提供数据方面的分析），目的是加强在健康数据分析方面的业务能力。随之而来的是，美国和加拿大的 14 家肿瘤中心将部署沃森（Watson）计算机系统，根据患者的肿瘤基因选择适当的治疗方案。IBM 与在线心理治疗公司 Talkspace 合作，通过机器学习，结合自然语言处理和用户个性分析技术，辅助用户决策，并帮助医生给出最佳治疗方案。

还是在 2015 年，IBM 宣布以 10 亿美元的价格收购医疗影像 Merge Healthcare 公司，并将其与新成立的 Watson Health 合并。Watson 不仅可以读懂这些医疗图像，还可以根据巨大的电子病历数据库进行分析诊断。通过收购 Merge Healthcare，IBM 增强了自身收集并传播影像的能力，而这正是通过机器学习诊断疾病的重要一步。紧接着，IBM 和美国第二大连锁药店 CVS 联合宣布，Watson 将和 CVS 一起，通过对相关指标和用户行为的分析，来预测其健康状况。CVS 将向 Watson 开放海量患者行为信息、临床数据、购药数据和保险数据等。通过对用户医疗健康记录、药店数据等信息的分析，可以预测用户患有疾病的风险，并向用户提供执业护士、医生以及相关的医疗保险等信息，为用户制定一个最佳的健康问题解决方案。这次合作将覆盖 CVS 的 7600 家连锁药店、1000 家医疗诊所。总辐射人群超过 7000 万人，占美国总人口的 22%。目前双方将第一阶段的合作放在慢病领域，包括高血压、心脏病、糖尿病和肥胖。这是医疗领域的一块最主要的市场（在美国，全年国民医疗总花费的 86%来源于慢病治疗）。

还是在 2015 年！2015 年 10 月初，IBM 宣布组建新的业务部门，要给 Watson 技术进行大规模商业化。这个新组建的部门名为“Cognitive Business Solutions”（认知商务解决方案）。这个新部门将拥有 2000 名员工，为企业提供有关如何利用 IBM 人工智能软件的咨询建议。从 IBM 的整个布局上看，机器学习在未来几年必然是非常火热的。

10.3.2　机器学习分类

机器学习按照学习形式进行分类，可分为监督学习、无监督学习、半监督学习和强化学习。我们先介绍几个术语。给定的数据集，可能包含以下几类字段：

- 特征（feature）：该类别描述了数据的属性，比如：年龄、教育、婚否、职业、收入、性别等都可能是某一个数据集的特征。这些特征可分为两大类：类别型特征（如：性别）和数值型特征（如：年龄）。
- 标签（label）：该类别描述了一些已知结果，其中结果与给定的一组特征有关。比如：风险类别就是标签。通过包含标签字段，我们使用算法找出一些特征与最终收入之间的关系。这个算法就是一个监督学习算法。
- 标识（identifier）：该类别描述了数据集中可以唯一标示数据的字段。

1. 监督学习

为了进行预测，需要从历史数据中获取大量的输入和相应的、已知正确的数据输出。整个输入输出数据集称为训练数据集。监督学习就是从给定的训练数据集中学习一个函数（模型），当新的数据到来时，可以根据这个函数（模型）预测结果。监督学习的训练集要求包括输入和输出，也可以说是特征和目标。训练集中的目标是由人标注（标量）的。在监督式学习下，输入数据被称为“训练数据”，每组训练数据有一个明确的标识或结果，比如对防垃圾邮件系统中“垃圾邮件”、“非垃圾邮件”，对手写数字识别中的“1”、“2”、“3”等。在建立预测模型时，监督式学习建立一个学习过程，将预测结果与“训练数据”的实际结果进行比较，不断调整预测模型，直到模型的预测结果达到一个预期的准确率。常见的监督学习算法包括回归分析和统计分类。回归问题的目标为数值型特征，而分类问题的目标为类别型特征。二元分类是机器学习要解决的基本问题，将测试数据分成两个类，比如垃圾邮件的判别、房贷是否允许等问题的判断。多元分类是二元分类的逻辑延伸。例如，在因特网的流分类的情况下，根据问题的分类，网页可以被归类为体育、新闻、技术等，依此类推。

监督学习常常用于分类，因为目标往往是让计算机去学习我们已经创建好的分类系统。数字识别再一次成为分类学习的常见样本。一般来说，对于那些有用的分类系统和容易判断的分类系统，分类学习都适用。

监督学习是训练神经网络和决策树的最常见技术。神经网络和决策树技术高度依赖于事先确定的分类系统给出的信息。对于神经网络来说，分类系统用于判断网络的错误，然后调整网络去适应它；对于决策树，分类系统用来判断哪些属性提供了最多的信息，如此一来可以用它解决分类系统的问题。

2. 无监督学习

与监督学习相比，无监督学习的训练集没有预先标注好的分类标签。在无监督式学习中，数据并不被特别标识，学习模型是为了推断出数据的一些内在结构。常见的应用场景包括关联规则的学习以及聚类等。常见算法包括 Apriori 算法和 k-Means 算法。

非监督学习看起来非常困难：目标是我们不告诉计算机怎么做，而是让它（计算机）自己去学习怎样做一些事情。非监督学习一般有两种思路：第一种思路是在指导 Agent 时不为其指定明确的分类，而是在成功时采用某种形式的激励制度。需要注意的是，这类训练通常会置于决策问题的框架里，因为它的目标不是产生一个分类系统，而是做出最大回报的决定。这种思路很好地概括了现实世界，Agent 可以对那些正确的行为做出激励，并对其他的行为进行处罚。

因为无监督学习假定没有事先分类的样本，这在一些情况下会非常强大，例如，我们的分类方法可能并非最佳选择。在这方面一个突出的例子是人机围棋大战，计算机程序通过非监督学习自己一遍又一遍地玩这个围棋游戏，变得比最强的人类棋手还要出色。这些程序发现的一些原则甚至令围棋专家都感到惊讶，并且它们比那些使用预分类样本训练的围棋程序工作得更出色。

3. 半监督学习

半监督学习是介于监督学习与无监督学习之间一种机器学习方式，是模式识别和机器学习领

域研究的重点问题。它主要考虑如何利用少量的标注样本和大量的未标注样本进行训练和分类的问题。半监督学习对于减少标注代价，提高学习机器性能具有非常重大的实际意义。主要算法有五类：基于概率的算法，在现有监督算法基础上进行修改的方法，直接依赖于聚类假设的方法，基于多试图的方法，基于图的方法。在半监督学习方式下，输入数据部分被标识，部分没有被标识，这种学习模型可以用来进行预测，但是模型首先需要学习数据的内在结构，以便合理地组织数据来进行预测。应用场景包括分类和回归，算法包括一些对常用监督式学习算法的延伸，这些算法首先试图对未标识数据进行建模，在此基础上再对标识的数据进行预测，如图论推理算法（Graph Inference）或者拉普拉斯支持向量机（Laplacian SVM）等。半监督学习分类算法提出的时间比较短，还有许多方面没有更深入的研究。

4. 强化学习

强化学习通过观察来学习动作的完成，每个动作都会对环境有所影响，学习对象根据观察到的周围环境的反馈来做出判断。在这种学习模式下，输入数据作为对模型的反馈，不像监督模型那样，输入数据仅仅是作为一个检查模型对错的方式，在强化学习下，输入数据直接反馈到模型，模型必须对此立刻做出调整。常见的应用场景包括动态系统以及机器人控制等。常见算法包括 Q-Learning 以及时间差学习（Temporal difference learning）。

在企业数据应用的场景下，人们最常用的可能就是监督式学习和非监督式学习的模型。在图像识别等领域，由于存在大量的非标识的数据和少量的可标识数据，目前半监督式学习是一个很热的话题。而强化学习更多地应用在机器人控制及其他需要进行系统控制的领域。

10.3.3　机器学习算法

机器学习算法是由普通的算法演化而来。通过自动地从提供的数据中学习，它会让你的程序变得更“聪明”。我们以机器学习中的经典故事“挑芒果”来解释机器学习技术。假定你从市场上的芒果里随机地抽取一定的样品（训练数据）， 制作一张表格，上面记着每个芒果的物理属性，比如颜色、大小、形状、产地、卖家等等（这些称之为特征）。还记录下这个芒果甜不甜，是否多汁，是否成熟（输出变量）。你将这些数据提供给一个机器学习算法（分类算法/回归算法），然后它就会做出一个关于芒果的物理属性和它的质量之间关系的模型。下次你再去超市，只要测测那些芒果的特性（测试数据），然后将它输入一个机器学习算法。算法将根据之前计算出的模型来预测芒果是甜的、熟的、并且还是多汁的。该算法内部使用的规则其实就类似决策树。

机器学习算法的一个优势是你可以让你的算法随着时间越变越好（增强学习），当它读进更多的训练数据，它就会更加准确，并且在做了错误的预测之后自我修正。这就是所谓的机器学习。

机器学习首先被用在人工智能领域，用于提升机器的自我学习能力。最近几年机器学习才被用于大数据分析领域。大数据分析（数据挖掘）受到很多学科领域的影响，其中机器学习、统计

学无疑影响最大。简言之，对数据挖掘而言，机器学习和统计学提供数据分析技术。统计学界提供的很多技术通常都要在机器学习界进一步研究，变成有效的机器学习算法之后，才能再进入数据挖掘领域。从这个意义上说，统计学主要是通过机器学习来对数据挖掘发挥影响，而机器学习则是数据挖掘的支撑技术。从数据分析的角度来看，绝大多数数据挖掘技术都来自机器学习领域，但机器学习研究往往并不把海量数据作为处理对象，因此，数据挖掘要对算法进行改造，使得算法性能和空间占用达到实用的地步。同时，数据挖掘还有自身独特的内容，即关联分析。

目前机器学习被广泛应用于信用卡欺诈、医疗大数据分析、语音识别、人脸识别等领域。在这些领域，输入数据和输出结果的关系比较复杂，因此，需要给机器提供一些测试数据（trainning data），让机器自己学习到输入数据和输出结果之前的关联关系（模式识别）。根据算法的功能和形式的类似性，我们可以把算法分类，比如说基于树的算法、基于神经网络的算法等等。当然，机器学习的范围非常庞大，有些算法很难明确归类到某一类。而对于有些分类来说，同一分类的算法可以针对不同类型的问题，下面用一些相对比较容易理解的方式来分类一些主要的机器学习算法：

（1）构造条件概率

- 回归分析和统计分类;
- 人工神经网络;
- 决策树;
- 高斯过程回归;
- 线性判别分析;
- 最近邻居法;
- 支持向量机。

（2）通过再生模型构造概率密度函数

- 最大期望算法;
- graphical model：包括贝叶斯网和 Markov 随机场;
- Generative Topographic Mapping。

（3）近似推断技术

- 马尔可夫链蒙特卡罗方法;
- 变分法。

10.4 Spark MLib

Spark 在机器学习方面具有得天独厚的优势。机器学习算法一般都有很多个步骤迭代计算的

过程，机器学习的计算需要在多次迭代后，获得足够小的误差或者足够收敛才会停止。迭代时如果使用 Hadoop 的 MapReduce 计算框架，每次计算都要读/写磁盘以及任务的启动等工作，这会导致非常大的 I/O 和 CPU 消耗。而 Spark 基于内存的计算模型就非常擅长迭代计算，多个步骤计算直接在内存中完成，只有在必要时才会操作磁盘和网络，所以说 Spark 正是机器学习的理想的平台。

10.4.1 MLib 架构

MLlib（Machine Learnig lib） 是 Spark 对常用的机器学习算法的实现库，同时包括相关的测试和数据生成器。MLlib 目前支持 4 种常见的机器学习问题：分类、回归、聚类和协同过滤。

MLlib 基于 RDD，可以与 Spark SQL、GraphX、Spark Streaming 无缝集成，以 RDD 为基石，4 个子框架（Spark SQL、Spark Streaming、MLib 和 GraphX）可联手构建大数据系统。MLlib 是 MLBase 一部分。MLBase 通过边界定义，力图将 MLBase 打造成一个机器学习平台，让不了解机器学习的用户也能方便地使用 MLBase 这个工具来处理自己的数据。MLBase 分为四部分：MLlib、MLI、ML Optimizer 和 MLRuntime：

- ML Optimizer 会选择它认为最适合的已经在内部实现好了的机器学习算法和相关参数来处理用户输入的数据，并返回模型或别的帮助分析的结果；
- MLI 是一个进行特征抽取和高级 ML 编程抽象算法实现的 API 或平台；
- MLlib 是 Spark 实现一些常见的机器学习算法和实用程序，包括分类、回归、聚类、协同过滤、降维以及底层优化，该算法可扩充；
- MLRuntime 基于 Spark 计算框架，将 Spark 的分布式计算应用到机器学习领域。

Spark MLlib 架构主要包含三个部分：

- 底层基础：包括 Spark 的运行库、矩阵库和向量库；底层基础部分主要包括向量接口和矩阵接口，这两种接口都会使用 Scala 语言基于 Netlib 和 BLAS/LAPACK 开发的线性代数库 Breeze;
- 算法库：包含广义线性模型、推荐系统、聚类、决策树和评估的算法;
- 实用程序：包括测试数据的生成、外部数据的读入等功能。

10.4.2 MLib 算法库

本小节我们分析一些 Spark 中常用的机器学习算法。

1. 分类算法

MLib 支持二元分类、多元分类等多种算法。分类算法属于监督式学习，使用类标签已知的样本建立一个分类函数或分类模型、应用分类模型，能把数据库中的类标签未知的数据进行归类。分类在数据挖掘中是一项重要的任务，目前在商业上应用最多，常见的典型应用场景有流失

预测、精确营销、客户获取、个性偏好等。MLlib 目前支持分类算法有：逻辑回归、线性支持向量机、朴素贝叶斯和决策树。

实例：加载一个训练数据集，然后在训练集上执行训练算法，最后在所得模型上进行预测并计算训练误差。

```
    import org.apache.spark.SparkContext
    import org.apache.spark.mllib.classification.SVMWithSGD
    import org.apache.spark.mllib.regression.LabeledPoint

    // 加载和解析数据文件
    val data = sc.textFile("mllib/data/sample_svm_data.txt")
    val parsedData = data.map { line =>
      val parts = line.split(' ')
      LabeledPoint(parts(0).toDouble, parts.tail.map(x =>
x.toDouble).toArray)
    }

    // 设置迭代次数并进行训练，建立模型
    val numIterations = 20
    val model = SVMWithSGD.train(parsedData, numIterations) //默认采用 L2正
则化

    // 评估模型，统计分类错误的样本比例
    val labelAndPreds = parsedData.map { point =>
    val prediction = model.predict(point.features)
    (point.label, prediction)
    }
    val trainErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble
/ parsedData.count
    println("Training Error = " + trainErr)
```

2. 回归算法

回归算法属于监督式学习，每个个体都有一个与之相关联的实数标签，并且我们希望在给出用于表示这些实体的数值特征后，所预测出的标签值尽可能接近实际值。MLlib 目前支持回归算法有：线性回归、岭回归、Lasso 和决策树。

实例：加载一个训练数据集，将其解析为带标签点的 RDD，使用 LinearRegressionWithSGD 算法建立一个简单的线性模型来预测标签的值，最后计算均方差来评估预测值与实际值的吻合度。

```
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
import org.apache.spark.mllib.regression.LabeledPoint

// 加载和解析数据文件
val data = sc.textFile("mllib/data/ridge-data/lpsa.data")
val parsedData = data.map { line =>
  val parts = line.split(',')
  LabeledPoint(parts(0).toDouble,   parts(1).split('   ').map(x   =>
x.toDouble).toArray)
}

//设置迭代次数并进行训练
val numIterations = 20
val model = LinearRegressionWithSGD.train(parsedData, numIterations)

// 统计回归错误的样本比例
val valuesAndPreds = parsedData.map { point =>
val prediction = model.predict(point.features)
(point.label, prediction)
}

val  MSE  =  valuesAndPreds.map{  case(v,  p)  =>  math.pow((v  -  p),
2)}.reduce(_ + _)/valuesAndPreds.count
println("training Mean Squared Error = " + MSE)
```

3. 聚类算法

聚类算法属于非监督式学习，它是研究分类问题的一种统计分析方法。它通常被用于探索性的分析，是根据“物以类聚”的原理，将本身没有类别的样本聚集成不同的组，这样的一组数据对象的集合叫做簇，并且对每一个这样的簇进行描述的过程。它的目的是使得属于同一簇的样本之间应该彼此相似，而不同簇的样本应该足够不相似，常见的典型应用场景有客户细分、客户研究、市场细分、价值评估。MLlib 目前支持广泛使用的 K-Mmeans 聚类算法。

实例：导入训练数据集，使用 K-Means 对象来将数据聚类到两个类簇中，所需的类簇个数会被传递到算法中，然后计算集内均方差总和（WSSSE），可以通过增加类簇的个数 k 来减小误差。

```
import org.apache.spark.mllib.clustering.KMeans

// 加载和解析数据文件
val data = sc.textFile("kmeans_data.txt")
```

```
val parsedData = data.map( _.split(' ').map(_.toDouble))

// 设置迭代次数、类簇的个数
val numIterations = 20
val numClusters = 2
// 使用 k-means 算法将数据聚集成2个类
val clusters = KMeans.train(parsedData, numClusters, numIterations)

// 评估集群内计算平方误差的综合，统计聚类错误的样本比例
val WSSSE = clusters.computeCost(parsedData)
println("Within Set Sum of Squared Errors = " + WSSSE)
```

4. 协同过滤算法

协同过滤算法常被应用于推荐系统，这些技术旨在补充用户-商品关联矩阵中所缺失的部分。Spark 支持 ALS 推荐引擎算法。

实例：导入训练数据集，数据每一行由一个用户、一个商品和相应的评分组成。假设评分是显性的，使用默认的 ALS.train()方法，通过计算预测出的评分的均方差来评估这个推荐模型。

```
import org.apache.spark.mllib.recommendation.ALS
import org.apache.spark.mllib.recommendation.Rating

// 加载和解析数据文件
val data = sc.textFile("mllib/data/als/test.data")
val ratings = data.map(_.split(',') match {
case Array(user, item, rate) => Rating(user.toInt, item.toInt, rate.toDouble)
})

// 设置迭代次数
val numIterations = 20
// 使用 ALS 构建推荐模型
val model = ALS.train(ratings, 1, 20, 0.01)

// 对推荐模型进行评分
val usersProducts = ratings.map{ case Rating(user, product, rate) => (user, product)}
val predictions = model.predict(usersProducts).map{
case Rating(user, product, rate) => ((user, product), rate)
}
```

```
val ratesAndPreds = ratings.map{
case Rating(user, product, rate) => ((user, product), rate)
}.join(predictions)

val MSE = ratesAndPreds.map{
case ((user, product), (r1, r2)) => math.pow((r1- r2), 2)
}.reduce(_ + _)/ratesAndPreds.count
println("Mean Squared Error = " + MSE)
```

10.4.3　决策树

本节将对机器学习领域中经典的分类和回归算法——随机森林（Random Forests）进行介绍。随机森林算法是机器学习、计算机视觉等领域内应用极为广泛的一个算法，它不仅可以用来做分类，也可用来做回归，即预测，随机森林算法由多个决策树构成，相比于单个决策树算法，它分类、预测效果更好，不容易出现过度拟合的情况。

随机森林算法基于决策树。决策树是数据挖掘与机器学习领域中一种非常重要的分类器，算法通过训练数据来构建一棵用于分类的树，从而对未知数据进行高效分类。举个相亲的例子来说明什么是决策树，如何构建一个决策树，以及如何利用决策树进行分类。某相亲网站通过调查相亲历史数据发现，女孩在实际相亲时有如下表现：

序号	城市拥有房产	婚姻历史（离过婚、单身）	年收入（单位：万元）	见面（是、否）
1	是	单身	12	是
2	否	单身	15	是
3	是	离过婚	10	是
4	否	单身	18	是
5	是	离过婚	25	是
6	是	单身	50	是
7	否	离过婚	35	是
8	是	离过婚	40	是
9	否	单身	60	是
10	否	离过婚	17	否

通过上面的历史数据可以构建如图 10-2 所示的决策树：

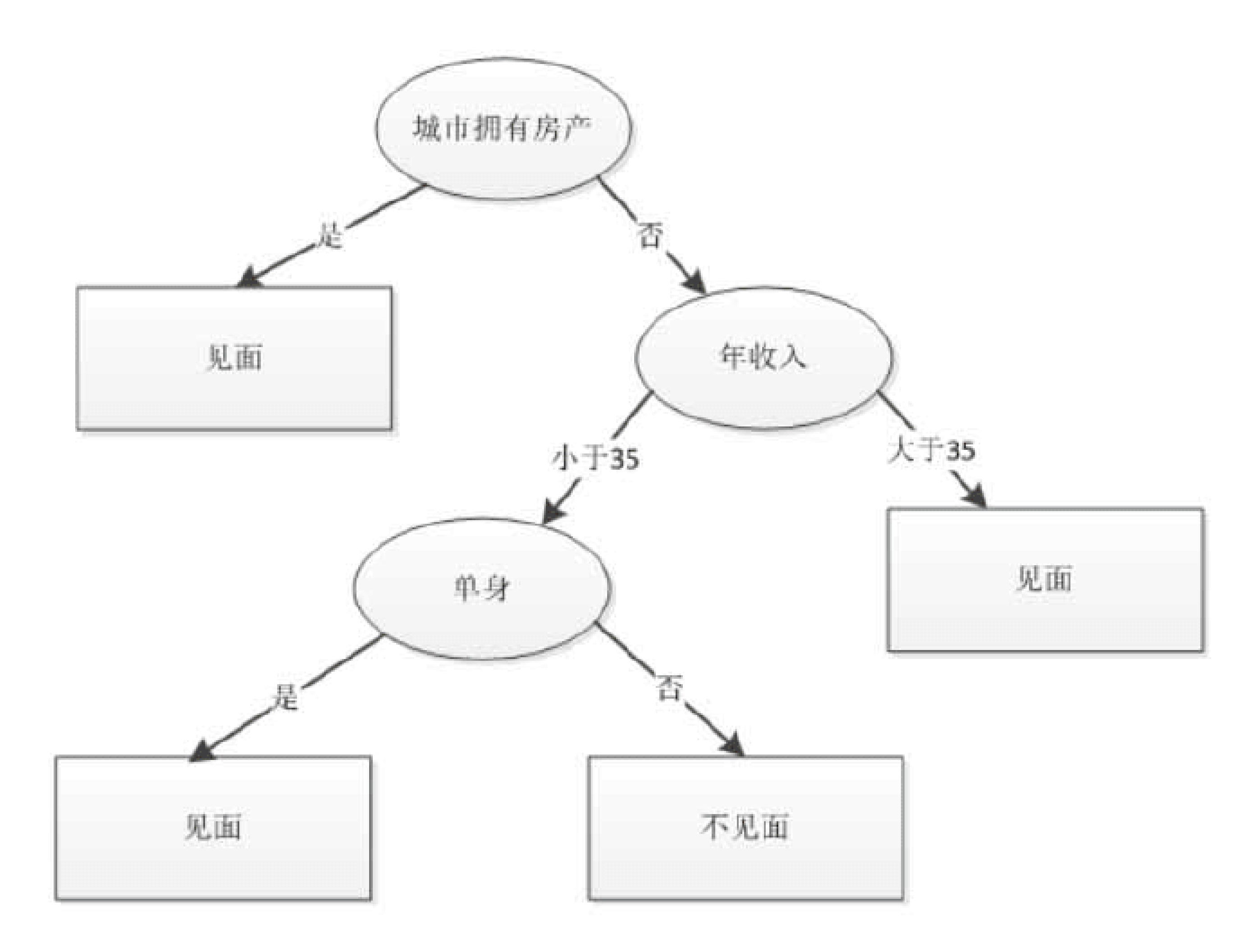


图 10-2 决策树

如果网站新注册了一个用户，他在城市无房产、年收入小于 35w 且离过婚，则可以预测女孩不会跟他见面。通过上面这个简单的例子可以看出，决策树对于现实生活具有很强的指导意义。通过该例子，我们也可以总结出决策树的构建步骤：

步骤01 将所有记录看作是一个节点。

步骤02 遍历每个变量的每种分割方式，找到最好的分割点。

步骤03 利用分割点将记录分割成两个子结点 C1 和 C2。

步骤04 对子结点 C1 和 C2 重复执行步骤（2）、（3），直到满足特定条件为止。

在构建决策树的过程中，最重要的是如何找到最好的分割点，那怎样的分割点才算是最好的呢？如果一个分割点能够将整个记录准确地分为两类，那该分割点就可以认为是最好的，此时被分成的两类是相对来说是最“纯”的。例如前面的相亲例子中，“在城市拥有房产”可以将所有记录分两类，所有是“是”的都可以划为一类，而“否”的则都被划为另外一类。所有“是”划分后的类是最“纯”的，因为所有在城市拥有房产单身男士，不管他是否离过婚、年收入多少都会见面；而所有“否”划分后的类，又被分为两类，其中有见面的，也有不见面的，因此它不是很纯，但对于整体记录来讲，它是最纯的。在上述例子当中，可以看到决策树既可以处理连续型变量也可以处理名称型变量。连续型变量如年收入，它可以用“>=”，“>”,“<”或“<=”作为分割条件；而名称型变量如城市是否拥有房产，值是有限的集合如“是”、“否“两种，它采用“=”作为分割条件。

在前面提到，寻找最好的分割点是通过量化分割后类的纯度来确定的，目前有三种纯度计算方式，分别是 Gini 不纯度、熵（Entropy）及错误率。关于这些计算方式的更多内容，读者可参考算法的专业书籍。

决策树算法负责为每层生成可能的决策规则。对于数值型特征，决策采用特征>=值的形式，对于类别型特征，决策采用特征在（值 1，值 2，…）中的形式。因此，要尝试的决策规则集合实际是可以嵌入决策规则中的一系列值。Spark MLib 把决策规则集合称为“bin”。bin 的数

量越多，需要的处理时间越多，但找到的决策规则可能更优。

决策树的构建是一个递归的过程，理想情况下所有的记录都能被精确分类，即生成决策树叶节点都有确定的类型，但现实中这种条件往往很难满足，这使得决策树在构建时可能很难停止。即使构建完成，也常常会使得最终的节点数过多，从而导致过度拟合（overfitting），因此在实际应用中需要设定停止条件，当达到停止条件时，直接停止决策树的构建。但这仍然不能完全解决过度拟合问题，过度拟合的典型表现是决策树对训练数据错误率很低，而对测试数据其错误率却非常高。过度拟合常见原因有：（1）训练数据中存在噪声；（2）数据不具有代表性。过度拟合的典型表现是决策树的节点过多，因此实际中常常需要对构建好的决策树进行枝叶裁剪（Prune Tree），但它不能解决根本问题，随机森林算法的出现能够较好地解决过度拟合问题。

随机森林算法由多个决策树构成的森林，算法分类结果由这些决策树投票得到，决策树在生成的过程当中，分别在行方向和列方向上添加随机过程，行方向上构建决策树时采用放回抽样（bootstraping）得到训练数据，列方向上采用无放回随机抽样得到特征子集，并据此得到其最优切分点，这便是随机森林算法的基本原理。随机森林是一个组合模型，内部仍然是基于决策树，同单一的决策树分类不同的是，随机森林通过多个决策树投票结果进行分类，算法不容易出现过度拟合问题。

随机森林算法在单机环境下很容易实现，但在分布式环境下特别是在 Spark 平台上，传统单机形式的迭代方式必须进行相应改进才能适用于分布式环境，这是因为在分布式环境下，数据也是分布式的，算法设计不得当会生成大量的 IO 操作，例如频繁的网络数据传输，从而影响算法效率。

下面是 Spark 官网上的随机森林算法的 demo 程序：

```
    import org.apache.spark.mllib.tree.RandomForest
    import org.apache.spark.mllib.tree.model.RandomForestModel
    import org.apache.spark.mllib.util.MLUtils
    // 加载数据
    val data = MLUtils.loadLibSVMFile(sc,
"data/mllib/sample_libsvm_data.txt")
    // 将数据随机分配为两份，一份用于训练，一份用于测试
    val splits = data.randomSplit(Array(0.7, 0.3))
    val (trainingData, testData) = (splits(0), splits(1))
    // 随机森林训练参数设置
    //分类数
    val numClasses = 2
    // categoricalFeaturesInfo 为空，意味着所有的特征为连续型变量
    val categoricalFeaturesInfo = Map[Int, Int]()
    //树的个数
    val numTrees = 3
    //特征子集采样策略，auto 表示算法自主选取
```

```
    val featureSubsetStrategy = "auto"
    //纯度计算
    val impurity = "gini"
    //树的最大层次，限制层数有利于避免对训练数据产生过拟合
    val maxDepth = 4
    //特征最大装箱数，对于连续的特征，其实就是进行范围划分，而划分的点就是 split（切分
    //点），划分出的区间就是 bin。
    val maxBins = 32
    //训练随机森林分类器，trainClassifier 返回的是 RandomForestModel 对象
    val model = RandomForest.trainClassifier(trainingData, numClasses,
categoricalFeaturesInfo,
     numTrees, featureSubsetStrategy, impurity, maxDepth, maxBins)

    // 测试数据评价训练好的分类器并计算错误率
    val labelAndPreds = testData.map { point =>
     val prediction = model.predict(point.features)
     (point.label, prediction)
    }
    val testErr = labelAndPreds.filter(r => r._1 != r._2).count.toDouble
/ testData.count()
    println("Test Error = " + testErr)
    println("Learned classification forest model:\n" +
model.toDebugString)

    // 将训练后的随机森林模型持久化
    model.save(sc, "myModelPath")
    //加载随机森林模型到内存
    val sameModel = RandomForestModel.load(sc, "myModelPath")
```

通过上述样例代码可以看到，从使用者的角度来看，随机森林中关键的类是 org.apache.spark.mllib.tree.RandomForest 和 org.apache.spark.mllib.tree.model.RandomForestModel 这两个类，它们提供了随机森林具体的 trainClassifier 和 predict 函数。

我们再来看一个案例来说明随机森林的具体应用。一般银行在货款之前都需要对客户的还款能力进行评估，但如果客户数据量比较庞大，信贷审核人员的压力会非常大，此时常常会希望通过计算机来进行辅助决策。随机森林算法可以在该场景下使用，例如可以将原有的历史数据输入到随机森林算法当中进行数据训练，利用训练后得到的模型对新的客户数据进行分类，这样便可以过滤掉大量的无还款能力的客户，如此便能极大地减少信贷审核人员的工作量。

记录号	是否拥有房产	婚姻情况	年收入（单位：万元）	是否具备还款能力

	（是/否）	（单身、已婚、离婚）		（是、否）
10001	否	已婚	10	是
10002	否	单身	8	是
10003	是	单身	13	是
……	…	…	…	……
11000	是	单身	8	否

将表中所有数据转换后，保存为 sample_data.txt，该数据用于训练随机森林。我们将测试数据保存为 input.txt。代码如下：

```
    object RandomForstExample {
     def main(args: Array[String]) {
     val sparkConf = new SparkConf().setAppName("RandomForestExample").
           setMaster("spark://sparkmaster:7077")
     val sc = new SparkContext(sparkConf)
     val      data:     RDD[LabeledPoint]     =     MLUtils.loadLibSVMFile(sc,
"/data/sample_data.txt")
     val numClasses = 2
     val featureSubsetStrategy = "auto"
     val numTrees = 3
     val model: RandomForestModel =RandomForest.trainClassifier(
                    data,
Strategy.defaultStrategy("classification"),numTrees,
     featureSubsetStrategy,new java.util.Random().nextInt())
     val     input:     RDD[LabeledPoint]     =     MLUtils.loadLibSVMFile(sc,
"/data/input.txt")
     val predictResult = input.map { point =>
     val prediction = model.predict(point.features)
     (point.label, prediction)
     }
     //打印输出结果，在 spark-shell 上执行时使用
     predictResult.collect()
     //将结果保存到 hdfs
     //predictResult.saveAsTextFile("/data/predictResult")
     sc.stop()
     }
    }
```

上述代码既可以打包后利用 spark-summit 提交到服务器上执行，也可以在 spark-shell 上执行查看结果。

10.5 深入了解算法

“如何分辨出垃圾邮件”“如何判断一笔交易是否属于欺诈”“如何判断一个细胞是否属于

肿瘤细胞”等等，这些问题都属于数据挖掘（Data Mining）的范畴，都很专业，都不太好回答。本节从数据挖掘的角度来深入了解算法，并通过现实中触手可及的、活生生的案例，去诠释它的真实存在。一般来说，数据挖掘的算法主要包含四种类型，即分类、预测、聚类、关联。前两种属于有监督学习，后两种属于无监督学习，属于描述性的模式识别和发现。有监督的学习，即存在目标变量，需要探索特征变量和目标变量之间的关系，在目标变量的监督下学习和优化算法。例如，信用评分模型就是典型的有监督学习，目标变量为“是否违约”。算法的目的在于研究特征变量（人口统计、资产属性等）和目标变量之间的关系。

无监督学习是指不存在目标变量，基于数据本身去识别变量之间内在的模式和特征。例如关联分析，通过数据发现项目 A 和项目 B 之间的关联性。例如聚类分析，通过距离将所有样本划分为几个稳定可区分的群体。这些都是在没有目标变量监督下的模式识别和分析。

10.5.1 分类算法

分类算法和预测算法的最大区别在于，前者的目标变量是分类离散型（例如，是否逾期、是否肿瘤细胞、是否垃圾邮件等），后者的目标变量是连续型。一般而言，具体的分类算法包括：逻辑回归、决策树、KNN、贝叶斯判别、SVM、随机森林、神经网络等。我们通过两个案例来深入了解分类算法。一个是垃圾邮件的分类和判断，另外一个是肿瘤细胞的判断和分辨。

1. 垃圾邮件的判别

邮箱系统如何分辨一封 Email 是否属于垃圾邮件？这应该属于文本挖掘的范畴，通常会采用朴素贝叶斯的方法进行判别。它的主要原理是，根据邮件正文中的单词，是否经常出现在垃圾邮件中进行判断。例如，如果一份邮件的正文中包含“报销”、“发票”、“促销”等词汇时，该邮件被判定为垃圾邮件的概率将会比较大。

一般来说，判断邮件是否属于垃圾邮件，应该包含以下几个步骤。

步骤01 把邮件正文拆解成单词组合，假设某篇邮件包含 100 个单词。

步骤02 根据贝叶斯条件概率，计算一封已经出现了这 100 个单词的邮件，属于垃圾邮件的概率和正常邮件的概率。如果结果表明，属于垃圾邮件的概率大于正常邮件的概率。那么该邮件就会被划为垃圾邮件。

2. 医学上的肿瘤判断

如何判断细胞是否属于肿瘤细胞呢？肿瘤细胞和普通细胞有差别，但是这需要非常有经验的医生，通过病理切片才能判断。如果通过机器学习的方式，使得系统自动识别出肿瘤细胞，此时的看病效率，将会得到飞速的提升。并且，通过主观（医生）+客观（模型）的方式识别肿瘤细胞，结果交叉验证，结论可能更加靠谱。那么，如何操作呢？通过分类模型识别。简言之，包含下面两个步骤。

步骤01 通过一系列指标刻画细胞特征，例如细胞的半径、质地、周长、面积、光滑度、对称

性、凹凸性等等，构成细胞特征的数据。

步骤 02 在细胞特征列表的基础上，通过搭建分类模型进行肿瘤细胞的判断。

10.5.2 预测算法

预测类算法，其目标变量一般是连续型变量。常见的算法，包括线性回归、回归树、神经网络、SVM 等。本小节主要介绍两个案例，即一个是通过化学特性判断和预测红酒的品质，另外一个是通过搜索引擎来预测和判断股价的波动和趋势。

1. 红酒品质的判断

如何评鉴红酒？有经验的人会说，红酒最重要的是口感。而口感的好坏，受很多因素的影响，例如年份、产地、气候、酿造的工艺等等。但是，统计学家并没有时间去品尝各种各样的红酒，他们觉得通过一些化学属性特征就能够很好地判断红酒的品质了。现在很多酿酒企业其实也都这么干了，通过监测红酒中化学成分的含量，从而控制红酒的品质和口感。那么，如何判断红酒的品质呢？

步骤 01 收集很多红酒样本，整理检测他们的化学特性，例如：酸性、含糖量、氯化物含量、硫含量、酒精度、PH 值、密度等等。

步骤 02 通过分类回归树模型进行预测和判断红酒的品质和等级。

2. 搜索引擎的搜索量和股价波动

Google 发现，互联网关键词的搜索量（例如流感）会比疾控中心提前 1~2 周预测出某地区流感的爆发。同样，现在也有些学者发现了这样一种现象，即公司在互联网中搜索量的变化，会显著影响公司股价的波动和趋势，即所谓的投资者注意力理论。该理论认为，公司在搜索引擎中的搜索量，代表了该股票被投资者关注的程度。因此，当一只股票的搜索频数增加时，说明投资者对该股票的关注度提升，从而使得该股票更容易被个人投资者购买，进一步地导致股票价格上升，带来正向的股票收益。这是已经得到无数论文验证了的。

10.5.3 聚类分析

聚类的目的就是实现对样本的细分，使得同组内的样本特征较为相似，不同组的样本特征差异较大。常见的聚类算法包括 Kmeans、系谱聚类、密度聚类等。对客户的细分，可以采用聚类分析。这能够有效地划分出客户群体，使得群体内部成员具有相似性，但是群体之间存在差异性。其目的在于识别不同的客户群体，然后针对不同的客户群体，精准地进行产品设计和推送，从而节约营销成本，提高营销效率。

例如，针对商业银行中的零售客户进行细分，基于零售客户的特征变量（人口特征、资产特征、负债特征、结算特征），计算客户之间的距离。然后，按照距离的远近，把相似的客户聚集为一类，从而有效地细分客户。将全体客户划分为诸如：理财偏好者、基金偏好者、活期偏好

者、国债偏好者、风险均衡者、渠道偏好者等。

Kmeans（K 均值聚类）是应用最广泛的聚类算法。它试图在数据集中找出 k 个簇群，这里 k 值由数据科学家指定。Spark MLib 提供了 Kmeans 的实现类，比如：

```
import org.apache.spark.mllib.clustering._
……
val kmeans = new KMeans()
kmeans.setK(k)
val model = kmeans.run(data)
model.clusterCenters.foreach(println)
```

上面这个代码首先建立了 KMeans 模型，然后输出每个簇的质心。

10.5.4　关联分析

关联分析的目的在于，找出项目（item）之间内在的联系。常常是指购物篮分析，即消费者常常会同时购买哪些产品（例如游泳裤、防晒霜），从而有助于商家的捆绑销售。

啤酒尿布是一个经典的关联分析的故事。故事是这样的，沃尔玛发现一个非常有趣的现象，即把尿布与啤酒这两种风马牛不相及的商品摆在一起，能够大幅增加两者的销量。原因在于，美国的妇女通常在家照顾孩子，所以，她们常常会嘱咐丈夫在下班回家的路上为孩子买尿布，而丈夫在买尿布的同时又会顺手购买自己爱喝的啤酒。沃尔玛从数据中发现了这种关联性，因此，将这两种商品并置，从而大大提高了关联销售。啤酒尿布主要讲的是产品之间的关联性，如果大量的数据表明，消费者购买 A 商品的同时，也会顺带着购买 B 产品。那么 A 和 B 之间存在关联性。在超市中，常常会看到两个商品的捆绑销售，很有可能就是关联分析的结果。

啤酒与尿布的故事很好地解释了数据挖掘中的关联规则挖掘的原理。我们也以这个故事来解释关联规则挖掘的基本概念。下面表 10-1 中的每一行代表一次购买清单（注意你购买十盒牛奶也只计一次，即只记录某个商品的出现与否）。数据记录的所有项的集合称为总项集，表中的总项集 S={牛奶,面包,尿布,啤酒,鸡蛋,可乐}。

表 10-1　某时刻商品关联关系表

时间（TID）	商品（Items）
T1	{牛奶,面包}
T2	{面包,尿布,啤酒,鸡蛋}
T3	{牛奶,尿布,啤酒,可乐}
T4	{面包,牛奶,尿布,啤酒}
T5	{面包,牛奶,尿布,可乐}

1. 关联规则、自信度、自持度的定义

关联规则就是有关联的规则，形式是这样定义的：两个不相交的非空集合 X、Y，如果有

X–>Y，就说 X–>Y 是一条关联规则。举个例子，在上面的表中，我们发现购买啤酒就一定会购买尿布，{啤酒}–>{尿布}就是一条关联规则。关联规则的强度用支持度（support）和自信度（confidence）来描述。

支持度的定义：support(X–>Y) = |X 交 Y|/N=集合 X 与集合 Y 中的项在一条记录中同时出现的次数/数据记录的个数。例如：support({啤酒}–>{尿布}) = 啤酒和尿布同时出现的次数/数据记录数 =3/5= 60%。

自信度的定义：confidence(X–>Y) = |X 交 Y|/|X| = 集合 X 与集合 Y 中的项在一条记录中同时出现的次数/集合 X 出现的个数 。例如：confidence({啤酒}–>{尿布}) = 啤酒和尿布同时出现的次数/啤酒出现的次数 = 3/3 = 100%。confidence({尿布}–>{啤酒}) = 啤酒和尿布同时出现的次数/尿布出现的次数 = 3/4 = 75%。

这里定义的支持度和自信度都是相对的支持度和自信度，不是绝对支持度，绝对支持度 abs_support = 数据记录数 N*support。支持度和自信度越高，说明规则越强，关联规则挖掘就是挖掘出满足一定强度的规则。

2. 关联规则挖掘的定义与步骤

关联规则挖掘的定义：给定一个交易数据集 T，找出其中所有支持度 support >= min_support、自信度 confidence >= min_confidence 的关联规则。

有一个简单的方法可以找出所需要的规则，那就是穷举项集的所有组合，并测试每个组合是否满足条件，一个元素个数为 n 的项集的组合个数为 2^n-1（除去空集），所需要的时间复杂度明显为 O(2^N)，对于普通的超市，其商品的项集数在 1 万以上，用指数时间复杂度的算法不能在可接受的时间内解决问题。怎样快速挖出满足条件的关联规则是关联挖掘的需要解决的主要问题。仔细想一下，我们会发现对于{啤酒–>尿布}、{尿布–>啤酒}这两个规则的支持度实际上只需要计算{尿布，啤酒}的支持度，即它们交集的支持度。于是我们把关联规则挖掘分两步进行：

步骤 01 生成频繁项集：这一阶段找出所有满足最小支持度的项集，找出的这些项集称为频繁项集。

步骤 02 生成规则：在上一步产生的频繁项集的基础上，生成满足最小自信度的规则，产生的规则称为强规则。

关联规则挖掘所花费的时间主要是在生成频繁项集上，因为找出的频繁项集往往不会很多，利用频繁项集生成规则也就不会花太多的时间，而生成频繁项集需要测试很多的备选项集，如果不加优化，所需的时间是 O(2^N)。

3. Apriori 定律

为了减少频繁项集的生成时间，我们应该尽早地消除一些完全不可能是频繁项集的集合，Apriori 的两条定律就是干这事的：

- Apriori 定律 1：如果一个集合是频繁项集，则它的所有子集都是频繁项集。举例：假设一个集合{A,B}是频繁项集，即 A、B 同时出现在一条记录的次数大于等于最小支持

度 min_support，则它的子集{A}、{B}出现次数必定大于等于 min_support，即它的子集都是频繁项集。

- Apriori 定律 2：如果一个集合不是频繁项集，则它的所有超集都不是频繁项集。举例：假设集合{A}不是频繁项集，即 A 出现的次数小于 min_support，则它的任何超集如{A,B}出现的次数必定小于 min_support，因此其超集必定也不是频繁项集。

利用这两条定律，我们抛掉很多的候选项集，Apriori 算法就是利用这两个定理来实现快速挖掘频繁项集的。

4. Apriori 算法

Apriori 算法是一个先验法，其实就是二级频繁项集是在一级频繁项集的基础上产生的，三级频繁项集是在二级频繁项集的基础上产生的，以此类推。Apriori 算法属于候选消除算法，是一个生成候选集、消除不满足条件的候选集、并不断循环直到不再产生候选集的过程。

5. FpGrowth 算法

Aprori 算法利用频繁集的两个特性，过滤了很多无关的集合，效率提高不少，但是我们发现 Apriori 算法是一个候选消除算法，每一次消除都需要扫描一次所有数据记录，造成整个算法在面临大数据集时显得无能为力。FpGrowth 算法效率就要比 Aprori 算法高很多。FpGrowth 算法通过构造一个树结构来压缩数据记录，使得挖掘频繁项集只需要扫描两次数据记录，而且该算法不需要生成候选集合，所以效率会比较高。

10.5.5 异常值分析算法

前面的四种算法类型（分类、预测、聚类、关联）是比较传统和常见的。还有其他一些比较有趣的算法分类和应用场景，例如协同过滤、异常值分析、社会网络、文本分析等。

基于异常值分析算法的一个案例就是支付中的交易欺诈侦测。当我们刷信用卡支付时，系统会实时判断这笔刷卡行为是否属于盗刷。通过判断刷卡的时间、地点、商户名称、金额、频率等要素进行判断。这里面基本的原理就是寻找异常值。如果您的刷卡被判定为异常，这笔交易可能会被终止。

异常值的判断，应该是基于一个欺诈规则库的。可能包含两类规则，即事件类规则和模型类规则：

- 事件类规则：例如刷卡的时间是否异常（凌晨刷卡）、刷卡的地点是否异常（非经常所在地刷卡）、刷卡的商户是否异常（被列入黑名单的套现商户）、刷卡金额是否异常（是否偏离正常均值的三倍标准差）、刷卡频次是否异常（高频密集刷卡）。
- 模型类规则：通过算法判定交易是否属于欺诈。一般通过支付数据、卖家数据、结算数据，构建模型进行分类问题的判断。

异常检测就是找出不寻常的情况。如果已经知道“异常”代表什么含义，我们就能通过监督

学习检测出数据集中的异常。在一些应用中，我们要能够找出以前从未见过的新型异常，如新欺诈方式。这些应用要用到非监督学习技术，通过学习，它们知道什么是正常输入，因此能够找出与历史数据有差异的新数据。这些新数据不一定是欺诈，它们只是不同寻常，因此值得我们做进一步调查。

10.5.6　协同过滤（推荐引擎）算法

基于协同过滤的案例之一就是电商猜你喜欢和推荐引擎。电商中的猜你喜欢，应该是大家最为熟悉的。在京东商城或者亚马逊购物，总会有“猜你喜欢”“根据您的浏览历史记录精心为您推荐”“购买此商品的顾客同时也购买了**商品”“浏览了该商品的顾客最终购买了**商品”，这些都是推荐引擎运算的结果。一般来说，电商的“猜你喜欢”（即推荐引擎）都是在协同过滤算法（Collaborative Filter）的基础上，搭建一套符合自身特点的规则库。即该算法会同时考虑其他顾客的选择和行为，在此基础上搭建产品相似性矩阵和用户相似性矩阵。基于此，找出最相似的顾客或最关联的产品，从而完成产品的推荐。我们再举一个例子。假定我们有一个数据集，它记录了用户和歌唱家（歌曲）之间的播放信息，其中包括播放的次数，但是数据集中没有包含用户和歌唱家的更多信息。那么，根据两个用户播放过许多相同歌曲来判断他们可能都喜欢某首歌，这叫做协同过滤。ALS 就是一个推荐引擎算法，Spark MLib 实现了 ALS 算法。

10.6　Mahout 简介

Mahout 起源于 2008 年，最初是 Apache Lucent 的子项目，它在极短的时间内取得了长足的发展，现在是 Apache 的顶级项目。Mahout 的主要目标是创建一些可扩展的机器学习领域经典算法的实现，旨在帮助开发人员更加方便快捷地创建智能应用程序。Mahout 现在已经包含了聚类、分类、推荐引擎（协同过滤）和频繁集挖掘等广泛使用的数据挖掘方法。除了算法，Mahout 还包含数据的输入/输出工具、与其他存储系统集成等数据挖掘支持架构。有兴趣的读者，可参考网站 https://mahout.apache.org/。

第 11 章

案例分析：环保大数据

环保行业的现状是：监控设备数量巨大、地理区域分散、实时性要求高；各个环保业务系统构成一个复杂的异构环境，这些系统之间的数据交换和共享几乎不太可能；与环境相关的数据不全面、不准确、不及时，并缺少对大量环境数据的统一整理；缺少对水体水源、大气、噪声、污染源、放射源、废气物等重点环保监测对象的状态、位置等信息进行全方位监控。

基于这些分析，环保行业需要分层设计的理念，分别在感知、传输、平台、应用 4 个层面进行信息化建设，以环保大数据管理平台为核心来构建和整合所有的智慧环保业务系统，确保所有环保数据和所有环保业务系统在一个统一的大数据管理平台上互联互通，从而为环保局提供一个综合的环保大数据应用平台，并为下一步的环保大数据分析奠定基础。

环保大数据应用平台是以环保大数据管理平台为基础，构建环境自动监测监控服务、综合查询服务、统计分析服务、视频监控服务、GIS 服务、应急服务、预警服务、电子政务服务、运营管理、移动应用、信访投诉、空气质量发布等环保业务子系统。整个平台集数据传输、采集、监控、数据统计、数据查询、趋势分析、决策支持、环境质量评价、污染预报、公共查询、数据上报、GIS 等功能为一体，结合各个地区已建成或将要建成的实时监测网，通过长期、连续、实时的数据分析，判断该地区的污染现状、污染趋势，评价污染控制措施的有效程度，研究污染对人们健康及对其他环境的危害，并为制定空气质量标准，验证污染扩散模式，以及进行污染预报，设计污染源的预警控制系统，制定经济有效的空气污染治理策略等提供依据。

11.1 环保大数据管理平台

大数据管理平台包含了环保大数据建模平台、环保大数据交换共享平台、环保大数据服务平台、环保大数据云平台四个核心子平台。环保相关的数据，如：环境质量检测数据、污染源数据、环境状况公报数据、城市考核数据、环境行政处罚数据、农村环境综合整治数据、环境统计数据、环境总量统计、污染物减排数据、放射源数据、环保应急预案等都在这个大平台上统一建模、统一采集和统一管理。大数据交换共享平台是一个可动态配置的环保物联中心，各类监控设备的数据通过交换平台进入大数据平台。在建模平台上统一定义所有的环境数据模型、环保业务流程模型、访问控制模型、存储模型等。数据服务平台提供了统一的接口来访问和操控平台上的

所有环境数据。

这个平台能够处理感知层通过传输层发送过来的海量监测数据的同时，也能将应用层发送的反控指令通过合理的渠道转发到感知层设备上，并且它能够存储 TB 级别的实时数据，根据实时数据的发展和历史状况，构建环保数据分析模型。这个平台能兼容原有系统，将原有的系统进行数据和业务上的整合。四个核心子平台的功能如下：

- 环保建模平台

提供数据建模工具，可对大气、水体水源、污染源、噪声、固危废、油烟、土壤等所有环保数据动态建模，数据模型中包括各类环保数据的访问控制列表、版本化管理、归档设置、是否加密、是否自动启动业务处理流程等。

提供 BPM 工具，可对所有环保业务流程统一建模。业务流程包含自动业务流程和人工干预业务流程。

- 环保数据交换共享平台

提供可配置的采集规则、异构数据源连接器、采集服务和采集管理控制台，实现了从多种异构系统和各类环保设备上自动采集环保相关数据。这些异构系统包括国家环保部和省环保厅下发的环保系统，市环保局现有的和未来要部署的环保系统，也包括与环保相关联的气象和交通等部门的信息系统。环保设备包括空气自动站、水站、污染源监测点位、噪声监测点位等相关设备发送的数据。

- 环保大数据云平台

海量存储市环保局历年来的环保数据和未来新增数据，支持单个表单可存储千亿行环保数据，总量 PB 级别。海量环保数据的查询速度在毫秒级别。提供了粒度到行和列的数据访问控制，支持多主节点多分节点的架构，横向可扩展至上百台节点（服务器）。

- 环保大数据服务平台

提供统一的接口和逻辑数据结构来访问和操控平台上所有环境数据，为环保数据的离线和在线分析和挖掘提供基于 Hive 和 Spark 的服务接口。提供对环境数据的申请、审批和接口开放等数据服务功能。预留与智慧城市大平台的接口，支持动态部署新的数据处理和分析应用。

11.2 环保大数据应用平台

环保大数据应用平台是以环保大数据管理平台为基础，构建环境自动监测监控服务、综合查询服务、统计分析服务、视频监控服务、GIS 服务、应急服务、预警服务、电子政务服务、运营管理服务、移动应用服务、信访投诉服务、空气质量发布服务等环保业务子系统。

11.2.1 环境自动监测监控服务

对所有污染源、环境质量监测点进行监控，实现对所有环境监测数据的全方位接入、组织、存储、管理和使用。以污染源监控为例，当污染源监测监控数据被采集到大数据平台之后，如下的污染源监控服务就开始监控：

1. 污染源总体监控信息

以列表的形式显示所有监测点的通信状况（在线、下线）、排放状况（正常、异常）、视频、基本信息、在线时间（当前、累积）、最新监测数据（数据超标变色）、污染源在线状况统计，按国控、省控、市控对污染源进行分类。

2. 污染源实时数据监测

对污染源监测数据进行实时监测，显示污染物浓度、流量实时曲线，实时表格，可单画面、多画面显示。可对多个监测点的数据进行对比检测，显示界面集成 GIS 和实时数据。

3. 治理设施过程监控

治理设施运行情况监测是通过实时采集和处理各种污染源在线监测仪表、治理设施和排污设备的关键参数，监测治理设施的运行状况和净化效果。关键参数包含电气参数（如电压、电流、频率参数）、工艺参数（物位、流量、压力等）。在设备上采用可靠的现场控制系统，监控治理设施的运行处理情况，同时，通过工厂总能源流转情况，在生产量估算的情况下，测算出污染排放量，以及应达到的净化指标，结合污染指标测算分析其综合治理情况，全面监测企业治理设施运行、污染物治理效果和排放量情况。

4. 噪声监控

在区域内的主要交通要道、学校、商业区和人口集中区域设置噪声自动监测和显示设备，对环境噪声进行 24×7 小时全天候实时监测，并通过电子显示屏向社会发布监测结果。市民可以随时看到自己居住附近或者途经交通干道的噪声分贝，直观了解噪声污染情况；各个测点的监测数据实时地传到数据中心，环保局用户可以对分布在区域内的各测点的数据进行实时监测，及时、准确地掌握噪声现状，分析其变化趋势和规律，了解各类噪声源的污染程度和范围，为城市噪声管理、治理和科学研究提供系统的监测资料。

5. 危险废物安全监控

利用 RFID 技术实现联单自动化处理。当危险废物运达处置单位时，RFID 射频识别设备通过发射信号自动识别目标对象（贴有 RFID 标签的危废）并获取相关数据（RFID 变迁存储的联单信息）。在读取到电子联单信息后，通过固废危废管理服务自动写入危废的种类名称、数量、产废单位、运输单位、承运人、运输起始时间、到达处置单位时间，危废处理方式等信息，并发送到相关负责人处审批。

运输监控管理服务主要结合运输车的 GPS 系统，对危废固废的运输路程、路线进行监控，

确保危废固废运输安全，不影响其他地区。运输车辆路线与原定路线出现偏差以后，系统将产生报警信息。点击出现报警情况的运输车辆，可以查看报警的详细信息。

6. 辐射监控

对各种辐射源进行数据指标的采样与收集。采集接入的辐射源监测点包括：环境自动监测点、III类以上工业放射源、城市放射性废物库、辐射环境监测标准子站。传输层把监测数据上传到数据中心。辐射安全监管服务完成对辐射源数据的处理和分析。

7. 报警监控

污染源报警监控服务包含：

- 报警查询：按类型快速查询报警信息，报警类型分为超标报警、数采仪掉线报警，超标报警可联动查看报警的视频图像或抓图。报警内容包括污染源名称、监测点名称、报警值、标准值、流量值、报警描述。查询条件包括时间、流量，并可以导出报警信息。
- 报警统计：统计一个企业或监测点一段时间内报警次数和报警持续时间，并对报警次数和时间进行排序。
- 报警处理：针对具体的某一条报警信息进行处理，并给出处置意见。
- 报警设置：设置报警上下限，异常值上下限，数采仪掉线时间间隔报警设置。报警的数据类型可根据用户需求进行灵活设置，包括小时数据、分钟数据、实时数据、日数据。
- 报警方式设置：设置报警方式（短信、邮件、网页弹窗提示、声音）、报警通知人、通知时间、是否启用报警；对报警处置配置模块中的通知人，处理人按科室进行分组。超标报警发给企业负责人，数采仪掉线报警发给运维单位相关人员。
- 送达报告：查看报警短信的送达情况报告。

8. 在线设备监控

- 实时地监控系统的各个设备和系统点的使用情况，及时地获知设备和系统的故障点。对设备的联网率、设备运行时间、排放状况、点位个数进行实时统计，反映设备整体的运行状况。
- 根据用户选择不同的排口，在界面上通过流程图和数据结合的方式显示出该测点设备在一天之中的运行时间、设备运行状况及实时数据。
- 除了监控管理之外，还包括故障管理、性能管理、安全管理和基础维护管理。

9. GPS 监控服务

通过 GPS，可以对收运车辆路线提供实时追踪服务。车载终端的 GPS 模块实时接收全球定位卫星的位置、时间等数据，一方面发送车内的监控系统，得到车辆的当前位置并且在电子地图上显示；另一方面，数据将通过 GPRS 终端模块发送到远程监控中心服务器上的大数据平台，

使得监控中心实时得到所有车辆的位置信息，给车辆的安全监控提供了基础。

10. 综合监控服务

比如，在大气监控中，运用物联网技术建立全空间（高空、近地和地面）全天候的三维大气监测体系，全面说清污染源状况及环境质量状况，全面跟踪工厂等主要污染源污染排放，及时掌握污染源状况以及实现污染源对污染浓度影响的技术分析，为大气污染应急提供决策手段，还可为大气污染防治措施、政策、标准等实施可能产生的效果进行科学评价，从而为大气环境管理提供科学决策能力。

11.2.2 综合查询服务

综合查询服务包括多个方面。查询结果可导出到 Excel、PDF 等格式的文档中。导出的报表抬头显示企业或监测点位名称。

1. 环境信息查询

按需实时查询其所关注的环境信息，例如：

- 重点污染源地理位置及其基本信息和相关环境信息。
- 某个区域内每日的空气污染指数。
- 某个区域内各类河流断面、湖库水质自动监测点等数据。
- 某个区域内空气质量自动监测站数据。
- 某个区域内饮用水源地保护区分布与保护范围。
- 自然保护区分布与保护内容。

通过与地理信息系统（GIS）的集成，可定位各类环境管理对象的地理位置，并可进行导航。另外，环保人员都可以通过移动智能终端进行查询。

2. 污染源数据查询

查询废气、污水流量、污染物浓度数据。数据类型分为实时数据、十分钟数据、小时数据、日数据。有时也需要完成综合查询，比如：按国控、省控、市控，日期查询整个地区或行业的废水、废气及污染物的排放量。

11.2.3 统计分析服务

按环境要素、业务功能需求分项统计各类报表内容，并对环境监测审核后的数据统计分析，生成 Excel、PDF 报表。还能够生成各类统计生产的季、月报表，对保存的季、月报表按用户需求再次统计。

在统计过程中，必然有一些判断的标准。若一个废水企业有多个排污口，可计算这几个企业排污口的污染物的平均值，并判断平均值是否达标；若是这个企业的多个排污口中有一个排污口

超标，就判断该企业废水超标。若一个废气企业有多个排污口，计算这几个企业排污口的污染物的平均值，并判断平均值是否达标；在多个排污口中只要一个排污口超标，就判断该企业废气超标。若一个企业既有废水排放口又有废气排污口，则按照上面的功能要求分别计算废水和废气的企业超标情况，并且如果废水和废气有一个排污口超标，则判断该企业超标。

除了常用的统计报表之外，还包括：

- 水环境质量统计报告：对主要河流，主要水库与湖泊、地下水质量进行汇总统计。
- 空气质量报告：包括按区域、污染指数、首要污染物、质量级别统计空气质量状况。
- 辐射环境质量分析报告：根据辐射监测数据进行汇总统计分析，反应辐射环境质量情况。

统计报表中包含单点及多个测点监测数据平均值计算，主要包括日、月、年平均值计算：

- 统计区域范围内监测数据最大值、最小值。
- 统计计算多个监测数据的标准差。
- 同期数据比较：对监测数据进行历史同期对比，以列表和对比图两种形式展示，进行辐射环境变化的趋势分析。
- 基准值比较：把监测数据与选定或输入的基准值进行比较，反映环境的优劣。
- 变化率计算：计算监测数据的月和年变化率。
- 自动监测与人工监测数据比较：设置人工监测数据输入接口，在同一图表中绘制自动监测结果与人工监测结果。
- 监测结果与本地对照分析：以列表和统计图两种方式展现监测数据与本地对照结果，并可对应显示在环境地理信息系统的电子地图上。

我们提供了多种分析服务。比如：环境质量分析。环境质量分析是利用现有环境监测数据，结合环境评估模型对环境质量进行分析，环境质量包括水环境质量、空气质量、声环境质量、辐射环境等。环境质量分析需要对各项环境进行独立分析，获取各区域的各类环境要素质量状况，为环境相关规定决策提供直接依据。

1. 水环境质量分析

利用各项监测数据实现对水环境质量进行分析，包括：河流、湖泊、水库、饮用水源地等水环境质量的监测和环境质量变化情况之间的关系进行分析。通过一定的数理方法与手段，对某一水环境区域进行环境要素分析，对其做出定量描述通过水环境质量评价，摸清区域水环境质量发展趋势及其变化规律，为区域环境系统的污染控制规划及区域环境系统工程方案的定制提供依据。水环境质量分析内容包括：水环境质量现状评估（根据各项监测数据对水环境质量现状进行评估），计算各大水系流域的水质类别并进行分布评估和分布对比，为环境质量的治理改善提供依据。

2. 空气质量分析

利用各项监测数据，分析计算区域内总体大气环境空气质量，以图和数据列表相结合的方式进行直观表达。可自动计算各区域的 API 指数，并分析 API 指数与污染因子之间的关系。

11.2.4 GIS 服务

实现各项环境业务数据和实际地理信息的有效结合和关联，可根据需要输出各种专题图，如：污染源分布图、大气质量功能区划图等。环境地理信息服务是为平台的各个业务提高地图服务和空间分析能力。在地图上能大致了解检测站点的位置时，可以通过地图放大功能将地图放大到监测站点的大致位置上，然后通过地图添加监测站点功能直接在地图上将监测站点信息添加到环保大数据平台上。我们提供了选择增加监测站点功能，可以将检测站点经纬度输入到监测站点信息内，系统自动定位监测站点到地图上，或者将地图放大到监测站点大致位置，点击将监测站点定位到地图上。

空间数据服务包括二部分：

- 空间数据管理：实现空间数据采集、编辑、入库、更新及存储和管理。
- GIS 服务管理：负责对空间数据资源、服务资源、接口资源的注册、发布、目录、安全进行管理。

在 GIS 服务上，它不仅可以向用户输出全要素地形图，而且可以根据用户需要分层输出各种专题图，如污染源分布图、大气质量功能区划图等等。在进行自然生态现状分析过程中，利用 GIS 可以比较精确地计算水土流失、荒漠化、森林砍伐面积等，客观地评价生态破坏程度和波及的范围，为各级政府进行生态环境综合治理提供科学依据。在环境影响评价时，对所有的改、扩、建项目可能产生的环境影响进行预测评价，并提供防止和减缓这种影响的对策与措施。利用 GIS 的空间分析服务，可以综合性地分析建设项目各种数据，帮助确立环境影响评价模型。利用 GIS 还可以更加明确地揭示不同区域的水环境状况，反映水体环境质量在空间上的变化趋势。可以更加直观地反映如污染源、排污口、监测断面等环境要素的空间分布。利用 GIS 还可以进行污染源预测、水质预测、水环境容量计算、污染物消减量的分配等，以表格和图形的方式为水环境管理决策提供多方位、多形式的支持。

11.2.5 视频服务

视频监控服务与污染源实时数据进行整合，在显示监测站点的视频图像的同时显示该站点的监测数据。

视频监控对危险流动源的收取运输处理的各个关键环节进行有效的实时监控，以确保对危险流动源收运过程的可视化的监控。

11.2.6　预警服务

通过物联网的监测信息，结合水环境、大气环境模拟模型，进行水、大气环境污染事故的预警分析。按预警源分为自动监测预警服务和人工预警服务。自动监测预警主要是从测控体系的子系统提取预警事件进行处理。人工预警监测主要从若干人工预警监测站点获取预警信息，系统提供这些人工监测站点数据录入、统计、分析功能。人工监测数据与自动监测数据一并进入预警系统。当与之有关的区域将要发生事故时，能提前发出预警，以便及时采取措施，防止事故的发生。

按照预警对象，分为：

- 水源水质预警

采用连续测定的仪器进行检测，运用 GIS 平台进行数据处理预测预报，一旦发现水质问题，向相关部门发送预警报告及相关处理方案，使得污染水体能够及时得到解决。

- 空气质量预警

利用污染源在线监测和空气质量在线监测，结合空气质量模型进行空气质量预警，并向相关环境管理人员进行汇报。

按照服务的方式，预警服务分为：

- 预警发布

当系统收到紧急预警或重要预警，管理员可以手工发布到预先设定的管理人员，发布方式有短信、网站公告等。

- 预警更改与解除

管理人员确认预警后，可以更改预警或解除预警。

- 预警查询

提供预警浏览界面，显示预警来源、时间、预警级别、预警内容等。查询功能提供查询界面，可以根据预警来源、时间、级别等查询条件查询相应预警。

- 预警指标管理

提供预警指标库的维护，包括新增预警、修改、删除、预警下发。

- 预警分级核定

提供预警分级核定功能。

11.2.7　应急服务

应急服务的最终目标是：构建环境质量预测及环境污染事故应急管理服务框架，加强环境污染应急处置及预案管理，提升应急反应和处置能力；对发生的环境事故，实现应急资源的调度和管理。应急服务包括突发事件应急处置指挥服务和环境安全应急指挥联动服务等。

我们通过“事件”来管理突发事故。对整个事件从产生、应急到处置进行闭环管理。减少管理的盲目性，提高监管效率。事件管理服务主要有如下功能：

- 事件分类管理
- 事件处理
- 事件分析
- 事件归档
- 事件浏览
- 事件管理

除了事件服务之外，还有如下服务：

1. 应急物资管理

针对可能出现的各种应急事故，对区域内各个地方存储应急物资如：灭火器、盐酸、消防栓、防毒面具进行统一管理。该服务详细描述了物资的用途、数量、存储地、负责人及联系方式，当出现应急事故时，指挥人员能及时调动相应物资处理。

2. 专家库管理

专家库将专家按照专业、类别进行分类，并将该专家的单位、电话联系方式留档，当出现应急事故时，指挥人员能够在第一时间与专家进行联系，保证事故能够得到科学合理的处置。

3. 应急预案管理

应急预案分为环保局预案、检查预案、辐射预案、危管预案等。环保局预案可以分为总则、组织机构与职责、应急处置、应急保障、应急通信联络、应急终止。用户可以详细地查询各个步骤的详细内容；检查预案可以分为总则、分队编成和职责、各种保障、环境监察应急工作程序。用户可详细查询每个步骤的详细内容；辐射预案包括应急分队编成、应急启动、开进、现场器材开展与监测、应急措施。用户可以详细查询每个步骤的详细内容。

4. 应急档案管理

档案管理将整个环境突发事故从发生到应急监测、处置的全过程记录进环保大数据平台。信息至少应该包括环境事故的类型、发生地、处理方法、所用处理物资、影响范围等。

5. 应急报告

根据事故现场提交的人工监测数据以及整个时间的处理过程，系统能够自动生成环境事故处

理评估报告，报告既可以根据人工监测数据对事故进行定量分析，又可以根据事故的影响范围、影响人群以及相应的定量分析结果做出定性的分析。

6. 环境评估

损益评估利用环境突发事故模型，根据事故的类型、污染类型、事故持续的时间按照既定的模型对环境事故造成的损失进行初步评估。

11.2.8　电子政务服务

电子政务服务中心集成环保 OA、总量减排、项目审批、污染源管理、许可证管理、现场执法、环境监察业务、行政处罚、固废和危废转移、核与辐射管理、环境信访、综合办公等多个业务，通过专用的工作流引擎，将环保局相关业务以任务的形式驱动，实现对业务办理进行跟踪、督办及考核的一体化业务管理。系统使得污染源从产生开始，自动将相关信息转后续监管部门共享，并且后续的信息自动归聚到同一污染源，随时动态反映污染源状况；同时加强环境质量数据管理，在污染源排污状况与环境质量状况之间建立联系，从而达到通过对点源的管理改善宏观环境质量的目标。

1. 总量减排服务

总量减排服务主要包括环境容量分析、减排电子台账、总量核算、总量统计与分析以及减排文件管理 4 部分内容。

电子台账的主要内容包括城市污水处理厂、企事业单位工业废水治理工程（含清洁生产、中水回用等）、产业结构调整（关停的废水或废气排放企业）、燃煤电厂脱硫工程、非电企业二氧化硫脱硫工程、产业结构调整（关停小火电机组）、油改气工程目录等相关数据的采集及相关文件、企业照片的上传。

总量核算包括 COD、氨氮、$S0_2$、NO_X 的历年完成减排量、十三五任务量、完成年任务比例、完成十三五任务比例相关数据采集及核算。

总量统计及分析显示各个指标的历史数据变化曲线、计算公式，同时能够统计出环统数据和根据监测得出的监测数据，同核算总量形成对比和分析，并进行形象、直观地展示。

2. 建设项目管理服务

建设项目管理服务包括网上申报及审批管理、环境影响评价管理、环保“三同时”验收管理、建设项目审批管理和总量控制管理等。

网上申报及审批管理：包括网上咨询、企业网上申报、环评单位网上申报、评估中心评估、网上资料核实、转网下处理、项目核实及批文发放、环评单位管理等内容组成。

环境影响评价管理：按照环境影响评价法的规定，对项目的环境影响评价大纲进行管理，对环评的各项数据进行复核和管理，对环评实陈结果进行评估。

建设项目审批管理：包括建设项目审批、建设项目试生产以及建设项目验收管理等内容。

总量控制管理：在建设项目审批以及建设项目验收管理过程中对于总量指标进行批复，并且与总量减排管理系统进行集成管理。

3. 排污许可证管理服务

排污许可证管理基于总量控制管理，建立一个工业污染源基本情况库，录入现有排污许可证的污染源基本情况数据，获取污染源基本数据，系统功能包括建立工业污染源及申报、排放数据库，对已发许可证按区域、流域等对污染物总量进行统计，对新发许可证进行总量分析评估，为控制和削减污染提供依据。排污许可证管理主要包括排污许可证发放管理、排污许可证换证、排污许可证注销、排污许可证年审、排污许可证企业监督管理等。

4. 污染源档案管理服务

一个典型的污染源，其生命周期分三个部分：污染源产生、污染源日常管理（许可、执法、收费、处罚等）、污染源的注销。污染源档案管理服务主要是基于污染源的全生命周期的变化，对全局范围内的污染源进行集中管理的系统。

5. 行政处罚管理服务

行政处罚管理服务包括调查取证、立案管理、案件受理、行政处罚告知、申辩管理、听证告知管理、听证通知管理、听证笔录及听证报告、审议管理、行政处罚决定、行政处罚跟踪管理、案件执行情况管理、案件复议及诉讼、配合强制执行、结案等。

6. 环境信访管理

信访投诉管理包括：电话投诉登记台、环保信访登记台、任务办理台、投诉调查处理、复函、转办函、环保信访查询库、投诉详情查看、查询统计等。

任务办理台集中环保投诉各业务处理于一身，专门用来各业务员在其中处理各类业务，包括环保投诉调查处理、领导的审核及签批、环保投诉登记表的打印、任务办理轨迹查看等。

民众可在环保网站、微信和手机客户端完成投诉信息登记，投诉信息基于大数据平台分级分层流转处理。

7. 排污收费服务

根据标准的排污收费流程运转，实现了日常工作流程化、具体工作电子化。

8. 综合办公中心

在整合环保局现有的办公系统平台功能基础上，建设一套符合国家有关规定和标准、符合环保局自身业务特点的综合办公系统。通过使用先进的信息技术，实现局内行政工作流程化定制与管理，减轻工作人员的工作强度，减少工作人员的无效工作，提高工作人员的工作效率，提高行政工作的成效。

综合办公系统在局机关、下属单位之间实现办公信息交互和共享，实现内部办公事务流程化处理，流程实现可视化定制和管理。该系统主要起到如下作用：

- 提高工作效率：不用拿着各种文件、申请、单据在各科室跑来跑去，等候审批、签发、盖章，这些都可在网络上进行。
- 规范单位管理：把一些弹性太大不够规范的工作流程做得井然有序，比如：公文会签、计划日志等工作流程审批都可在网上进行。
- 使决策变得迅速科学：高层决策不再是在不了解情况、缺乏数据的环境下主管决策，而是以数据和真相为依据做出的科学决策。
- 提高内部凝聚力：科员与上级沟通很方便，信息反馈畅通，为发挥科员的智慧和积极性提供了舞台。

综合办公系统包括公文管理、会议管理、车辆管理、移动办公、领导日程管理、通信录管理、待办事宜、催办督办以及行政事务管理等内容。

11.2.9 智能化运营管理系统

运营管理系统是一个为环保局和运营公司使用的综合系统，提供了三大系统：在线监测系统、运营管理系统和运营管理后台系统，一共 25 个子系统。其中在线监测包括：统计分析、地理信息、现场端信息、实时数据、连接状态、运维统计和视频监控七个子系统；运营管理系统包括：个人主页、事件管理、人员管理、标准与规范、成本核算、绩效查看、环保知识库、FAQ 和环保专家库九个子系统；运营管理后台系统包括：新闻公告、企业基本信息、人员管理、任务发布、FAQ 论坛管理、资料文档上传、坐标查询工具、QR 码工具和绩效考核九个子系统。

11.2.10 环保移动应用系统

安装在环保业务人员的智能手机上的环保移动应用，访问大数据平台上的数据来进行数据查询和业务操作，实现移动办公和监控。环保局管理人员与执法人员能够实现信息的及时获取及传送，现场执法、现场办公以及信息的记录等功能，提高环境管理的效率和执法的准确性。

在环保移动应用上，环保人员可以查询各类数据（如：污染源地图、环境质量信息、办公信息、法律法规），完成现场执法、稽查管理等多个功能，同时将执法后的信息立即保存到环保大数据平台上。在平台上，专门为移动应用提供了相应的移动服务，手机应用通过这些服务访问和管理数据中心上的数据。还有，智能手机上的 GPS 定位环保人员的当前位置和行进轨迹。当有突发环境或信访事件发生时，便于指挥中心对车辆和人员进行调度，及时对事件进行处置。

现场执法人员还可通过环保移动应用在对环境违法企业进行现场执法。比如：记录问讯笔录、取证（拍照、摄像、录音等），以前需要手工填写问讯笔录，利用摄录设备取证。目前可以通过移动应用，利用智能手机完成笔录和取证工作，笔录和取证数据立即保存在环保大数据平台，提高了工作效率。

环保移动应用还具有污染源现场核查的功能。从前，环境监察部门巡检需要准备并携带大量的表格、文书、参考资料，到现场边检查边填表，对有疑惑的问题要现场翻阅相关法律法规条文和文档资料。检查完毕返回后再根据现场填写表格的进行计算和存档。通过环保移动应用，按照

设定流程填写完检查单。

还有，各类环保工作人员外出进行现场执法和检查或出差，其间可通过环保移动应用查询各类环境新闻，可查询环保部、省政府及环保厅下发的各类文件和通知、公告等，可进入移动办公系统进行文件办理。

为加强对各地污染企业自动监测数据管理，监控中心每月不定期按巡查比对、监督检查等要求对企业进行检查并且人工监测污染企业的污染数据。检查的项目包括：人工监测平台是否规范、排污单位是否违规进入站房、企业是否修改设备参数、企业是否模拟数据上传、数据上传是否符合要求、分析仪器数据是否与上传数据一致、是否存在虚报停产的情况、历史问题是否已经整改以及其他违反规定的情况等。对于违规企业以及拒绝检查人员检查的企业，按照有关文件规定，在计算当月运行率、准确率和超标情况时按照违反规定的程度来统计。另外，将人工监测数据与人工监测数据同时的自动监测数据相比较。在计算当月运行率、准确率和超标情况时按照人工监测数据与自动监测数据相差程度来统计。

当人工监测数据与自动监测数据的误差超过国家有关规定时，使用人工监测数据参与各种报表统计分析。实现人工监测数据与自动监测数据的对接。结合监控交流填报的企业生产状态，自动判断出历史问题是否整改，并作出相应处理。所有这些处理，都可以在环保移动应用上完成。

11.2.11　空气质量发布系统

本系统主要展示各个监测站 24 小时、48 小时、15 天和 30 天的实时数据和相关的趋势图。对监测点位的空气质量监测数据进行各个时段的实时数据和历史数据分析，显示二氧化硫、二氧化氮、可吸入颗粒物（PM_{10}）、$PM_{2.5}$ 实时数据。总体上能够体现民众所关心的空气质量信息。

11.3 环保大数据分析系统

环保大数据分析系统的目标是通过采用数据分析、可视化监测和地理信息技术，并通过环保专家知识库和对大量实时和历史数据的挖掘、评测与关联性分析，深度获取和挖掘相关环境数据，帮助环保部门准确判断环境变化趋势。同时把环保危机事件的预警、态势分析、联动和应急指挥决策辅助融为一体，提供准确的分析，挖掘掌握水、气、土壤等多项生态环境变迁和关联性的规律，对完善环境法律、法规体系、环保行业监测规程和技术标准、环保发展战略的规划等提供充分的科学依据。

第 12 章 案例分析：公安大数据

公安大数据是指基于大数据技术来满足公安数据的海量汇聚和智能分析的需要。通过部署公安大数据管理平台，为公安行业构建分布式处理运行环境和集群管理平台，建设公安行业统一的分布式数据资源层（池）。这个数据资源层为公安行业提供了如下功能：

- 集中、高效的数据资源管理体系和数据资源目录体系，强化公安数据标准管理，提升数据管理能力；
- 统一身份认证、数据访问控制和授权管理、数据审核机制，提升数据安全能力；
- 提供数据综合门户、数据资源服务系统、数据请求服务系统等功能，形成了统一的数据资源服务体系，为公安信息研判、业务应用、信息共享提供数据服务支撑环境，提升数据服务能力；
- 实现省市两级信息资源互通级联，不同业务系统之间互联互通，提升协作能力；
- 边整合，边应用，首先构建人员、机动车、案件专题数据库，为警务实战应用提供全要素、档案式资源支撑，提升创新能力；
- 实现公安数据资源采集、交换、服务、应用的全链路展示。

12.1 总体架构设计

基于分布式计算架构，公安大数据管理平台采用大数据分布式集群系统架构，提供分布式数据服务基础资源池，为分布式数据创新应用提供基础平台，是 “警务大数据”工程数据挖掘分析服务平台、智慧平台的基础保障系统，为警务大数据分析应用统一提供高性能的分布式计算、大数据处理和资源服务支撑。

大数据管理平台可细分为公安大数据建模平台、公安大数据交换共享平台、公安大数据云平台、公安大数据服务平台。

公安大数据交换共享平台负责公安数据的采集与交换，数据采集需包含传统数据、海量数据的采集。其中海量数据的采集要利用大数据相关技术。数据的交换也需包含传统数据交换、海量数据交换。

公安大数据云平台需利用分布式架构技术提供的分布式存储、分布式运算对海量数据根据公安业务的实际要求进行存储与运算。数据存储要保证安全、高效。数据运算要利用分布式架构所

提供的分布式运算，可以对海量的数据进行快速、高效运算以达到公安业务的需求。

公安大数据建模平台需完成以人建库、以车建库、以案建库的方式，实现人员大数据、车辆大数据、案件大数据。并对公安业务中固化的、常态的业务需要形成相对应的一系列主题库与专题库。

公安大数据服务平台提供数据（和数据目录）访问入口，分布式处理所形成的数据可以通过数据入口来满足各类警务应用系统对数据资源访问要求。还需提供对数据的管理入口，通过管理入口可以对大数据集成形成的数据做符合公安工作的数据管理。

12.2 建设内容

按照公安部《关于印发〈公安信息资源服务平台建设任务书〉的通知》的要求，大数据管理平台的主要建设内容如下：

- 建设分布式系统集群

部署“大数据”分布式处理运行环境和集群管理平台，为警务大数据应用提供高性能的分布式计算服务。

- 建设分布式数据资源层

统一建设全警共享的大数据资源池，实现分布式数据集成，构建人员、机动车、案件专题数据库，为警务实战应用提供全要素、档案式资源支撑。

- 建设分布式数据资源服务

强化公安数据标准管理和数据授权管理，实现省市两级信息资源服务总线互通级联，实现数据资源采集、交换、服务、应用的可视化、全链路展示。

各子平台主要功能列表如表 12-1 所示。

表 12-1　各子平台主要功能

子平台	子系统	主要功能
大数据云平台（分布式系统集群）	安装部署分布式集群环境	操作系统配置
		集群安装
	集群管理	硬件资源注册
		节点配置部署
		集群运行监控
		集群组件服务
		集群安全管理

（续表）

子平台	子系统	主要功能
大数据建模平台	数据建模	数据（标准）模型建立和管理
		标准检测管理
		标准变更申报与下发
		数据级联管理
		数据标准引用
		专题库（人、车、案）管理
	存储建模	数据存储管理
	安全建模	权限申请、权限管理
		角色管理
		角色授权管理
		权限分配查询
		统一用户和用户组管理
		访问控制管理
大数据交换共享平台	采集规则管理	异构数据源采集规则、映射规则，协议转换规则管理
	批量数据处理工具	资源上传管理
	连接器管理	连接异构数据源，并获取数据
	采集服务管理	数据资源采集监控
		数据交换监控
大数据服务平台	数据资源服务	分布式数据访问服务
		数据资源全链路展示
		数据版本化服务
	目录资源管理	分布式资源目录管理
	目录资源服务	请求资源目录
	数据申请管理	申请及批复管理
	监控中心	数据资源利用监控
		审计管理
		日志管理

12.3 建设步骤

公安大数据的建设步骤可简单分为以下 5 个步骤：

步骤01 部署大数据管理平台，包括部署分布式集群系统；

步骤02 配置数据模型和安全模型，完成数据级联、数据标准、数据授权等功能，配置采集规则和交换规则，实现数据资源采集和整合，初步形成分布式资源池。

步骤03 在大数据建模平台上完成人、车、案等各类专题库。

步骤04 通过大数据服务平台为大数据分析提供分布式计算服务、分布式数据处理、批量数据请求工具，实现数据资源全链路展示功能。

步骤05 开展公安大数据分析。

附录 1

◀ 数据量的单位级别 ▶

计算机存储最小的基本单位是 bit，按顺序给出所有单位：bit、Byte、KB、MB、GB、TB、PB、EB、ZB、YB、BB、NB、DB。它们按照进率 1024（2 的十次方）来计算：

```
8 bit = 1 Byte
1 KB = 1,024 Bytes
1 MB = 1,024 KB = 1,048,576 Bytes
1 GB = 1,024 MB = 1,048,576 KB
1 TB = 1,024 GB = 1,048,576 MB
1 PB = 1,024 TB = 1,048,576 GB
1 EB = 1,024 PB = 1,048,576 TB
1 ZB = 1,024 EB = 1,048,576 PB
1 YB = 1,024 ZB = 1,048,576 EB
1 BB = 1,024 YB = 1,048,576 ZB
1 NB = 1,024 BB = 1,048,576 YB
1 DB = 1,024 NB = 1,048,576 BB
```

附录 2

Linux Shell常见命令

Linux 系统为用户提供两个接口，一个是面向操作命令的接口 shell 和面向程序开发的接口 API。

Shell 是 Linux 系统中用户与内核进行交互操作的一种接口。它接收用户输入的命令并把它送入内核去执行。实际上 Shell 是一个命令解释器，它解释由用户输入的命令并且把它们送到内核。不仅如此，Shell 有自己的编程语言用于对命令的编辑，它允许用户编写由 shell 命令组成的程序。Shell 编程语言具有普通编程语言的很多特点，比如它也有循环结构和分支控制结构等，用这种编程语言编写的 Shell 程序与其他应用程序具有同样的效果。

每个 Linux 系统的用户可以拥有他自己的用户界面或 Shell，用以满足他们自己专门的 Shell 需要。Shell 也有多种不同的版本。主要有下列版本的 Shell。

- Bourne Shell：是贝尔实验室开发的。
- BASH：是 GNU 的 Bourne Again Shell，是 GNU 操作系统上默认的 shell。本书例子上使用的就是 BASH Shell。
- Korn Shell：是对 Bourne SHell 的发展，大部分内容与 Bourne Shell 兼容。
- C Shell：是 SUN 公司 Shell 的 BSD 版本。
- Z Shell：Z 是最后一个字母，也就是终极 Shell。它集成了 bash、ksh 的重要特性，同时又增加了自己独有的特性。

Shell 就是一个程序，它接受从键盘输入的命令， 然后把命令传递给操作系统去执行。

终端仿真器

当使用图形用户界面时，我们需要另一个和 shell 交互的叫做终端仿真器的程序。 我们单击右键，会找到一个 “terminal”。

第一次按键

启动终端仿真器，一旦它运行起来，我们应该看到一行像这样的文字：

```
[me@linuxbox ~]$
```

这叫做 shell 提示符，无论何时当 shell 准备好了去接受输入时，它就会出现。它可能会以各种各样的面孔显示，这取决于不同的 Linux 发行版，通常包括你的用户名@主机名，紧接着

当前工作目录（稍后会有更多介绍）和一个美元符号。

如果提示符的最后一个字符是“#”，而不是“$”，那么这个终端会话就有超级用户权限。这意味着，我们或者是以 root 用户的身份登录，或者是我们选择的终端仿真器提供超级用户（管理员）权限。

命令历史

如果按下上箭头按键，我们会看到刚才输入的命令重新出现在提示符之后。这就叫做命令历史。许多 Linux 发行版默认保存最后输入的 500 个命令。按下下箭头按键，先前输入的命令就消失了。

移动光标

可借助上箭头按键，来获得上次输入的命令。现在试着使用左右箭头按键。体会一下怎样把光标定位到命令行的任意位置。

关于鼠标和光标

虽然，shell 是和键盘打交道的，但你也可以在终端仿真器里使用鼠标。X 窗口系统内建了一种机制，支持快速拷贝和粘贴。如果你按下鼠标左键，沿着文本拖动鼠标（或者双击一个单词）高亮了一些文本，那么这些高亮的文本就被拷贝到了一个由 X 管理的缓冲区里面。然后按下鼠标中键，这些文本就被粘贴到光标所在的位置。

运行一些简单命令

现在，我们执行一些简单的命令。第一个命令是 date。这个命令显示系统当前的时间和日期。

```
[me@linuxbox ~]$ date
Thu Oct 25 13:51:54 EDT 2015
```

一个相关联的命令，cal，它默认显示当前月份的日历。

```
[me@linuxbox ~]$ cal
October 2015
Su Mo Tu We Th Fr Sa
1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

查看磁盘剩余空间的数量，输入 df。

```
[me@linuxbox ~]$ df
```

```
Filesystem          1K-blocks      Used Available Use% Mounted on
/dev/sda2            15115452   5012392   9949716  34% /
/dev/sda5            59631908  26545424  30008432  47% /home
/dev/sda1              147764     17370    122765  13% /boot
tmpfs                  256856         0    256856   0% /dev/shm
```

同样地，显示空闲内存的数量，输入命令 free。

```
[me@linuxbox ~]$ free
total       used       free     shared    buffers     cached
Mem:       2059676     846456    1213220          0
44028      360568
-/+ buffers/cache:     441860    1617816
Swap:      1042428          0    1042428
```

结束终端会话

我们可以通过关闭终端仿真器窗口，或者是在 shell 提示符下输入 exit 命令来终止一个终端会话：

```
[me@linuxbox ~]$ exit
```

附录 3

◀ Ganglia（分布式监控系统） ▶

系统部署上线之后，我们无法保证系统 7×24 小时都正常运行，即使是在运行着，我们也无法保证 Job 不堆积、是否及时处理 Kafka 中的数据。所以我们需要实时地监控系统，包括监控 Flume、Kafka 集群、Spark Streaming 程序。其中的一个监控系统就是 Ganglia，一旦检测到异常，系统会自己先重试是否可以自己恢复，如果不行，就会给我们发送报警邮件，甚至是打电话。

Ganglia 是 UC Berkeley 发起的一个开源集群监视项目，是一个跨平台、可扩展的、高性能计算系统下的分布式监控系统。它设计用于测量数以千计的节点。Ganglia 的核心包含下面三个组件：

- gmond（数据监测节点）：这个部件安装在要监测的节点上，用于收集节点的运行情况，并将这些统计信息发送到 gmetad 上；
- gmetad（数据收集节点）：该部件用于收集 gmond 发送的数据；
- 一个 Web 前端：用于将 gmetad 整理生成的 xml 数据以网页形式呈现给用户。

gmetad 可以部署在集群内任一台节点或者通过网络连接到集群的独立主机，它通过单播路由的方式与 gmond 通信，收集区域内节点的状态信息，并以 XML 数据的形式，保存在数据库中。由 RRDTool 工具处理数据，并生成相应的图形显示，以 Web 方式直观地提供给客户端。

Ganglia 主要是用来监控系统性能，如：cpu 、mem、硬盘利用率， I/O 负载、网络流量情况等，通过曲线很容易见到每个节点的工作状态，对合理调整、分配系统资源，提高系统整体性能起到重要作用。gmond 带来的系统负载非常少，这使得它成为在集群中各台计算机上运行的一段代码，而不会影响用户性能。

Ganglia 可以用于监控 Hadoop 项目，比如：Ganglia 可监控 HBase 相关进程的 Requests 和 Compactions Queue 等。

附录 4

◀ auth-ssh脚本 ▶

```
#!/bin/bash
#Title: auth-ssh.sh
#Description: no keys to ssh connecte each other
#system: linux
#Author:ZY
#Data:2015-01-05
#version 1.1

passwd =123456

if [ `rpm -qa | grep expect | wc -l` -eq 0 ]
then
    yum install expect -y
fi

for ip in `cat /etc/hosts | sed 1,2d | awk '{if(NF>0) print $1}'`
do
expect <<!
spawn ssh $ip rm -f /root/.ssh/*
  expect {
          "(yes/no)"  { send "yes\r"; exp_continue }
          "password:" { send "$passwd\r"; exp_continue }
        }
!
done

for ip in `cat /etc/hosts | sed 1,2d | awk '{if(NF>0) print $1}'`
do
expect <<!
spawn ssh $ip ssh-keygen -t rsa >/dev/null
```

```
  expect {
         "(yes/no)"  { send "yes\r"; exp_continue }
   "password:" { send "$passwd\r"; exp_continue }
   "(/root/.ssh/id_rsa):" { send "\r"; exp_continue }
   "(empty for no passphrase):" { send "\r"; exp_continue }
   "again:" { send "\r" }
 }
spawn scp $ip:~/.ssh/id_rsa.pub /root/
  expect {
         "(yes/no)"  { send "yes\r"; exp_continue }
         "password:" { send "$passwd\r"; exp_continue }
       }
!
cat /root/id_rsa.pub >> ~/.ssh/authorized_keys
rm -f /root/id_rsa.pub
done

for ip in `cat /etc/hosts | sed 1,2d | awk '{if(NF>0) print $1}'`
do
expect <<!
spawn scp /root/.ssh/authorized_keys $ip:~/.ssh/authorized_keys
  expect {
         "(yes/no)"  { send "yes\r"; exp_continue }
         "password:" { send "$passwd\r"; exp_continue }
       }
!
done
```

附录 5

◂ 作者简介 ▸

本书作者杨正洪是国内知名大数据专家，毕业于美国 State University of New York at Stony Brook，在 IBM 公司从事大数据相关研发工作 12 年多。从 2003~2013 年，杨正洪在美国加州的 IBM 硅谷实验室（IBM Silicon Valley Lab）负责 IBM 大数据平台的设计、研发和实施，主持了保险行业、金融行业、政府行业的大数据系统的架构设计和实施。

杨正洪是华中科技大学和中国地质大学客座教授，拥有国家专利，是湖北省 2013 年海外引进人才。受武汉市政府邀请，杨正洪于 2012 年 12 月发起成立武汉市云升科技发展有限公司，并获得东湖高新技术开发区办公场所和资金支持。目前公司在浙江和上海分别有全资子公司，在美国硅谷设有研发中心。公司的核心产品是大数据管理平台 EasyDoop，并以 EasyDoop 为基础研发了公安大数据产品和环保大数据产品。这些产品在公安和环保行业得到成功实施，三次被中央电视台新闻联播节目播报，省部长级政府领导亲自考察，并给予了很高的评价。

杨正洪参与了多项大数据相关标准的制定工作，曾受邀参与了公安部主导的“信息安全技术-大数据平台安全管理产品安全技术要求”的国家标准制定。